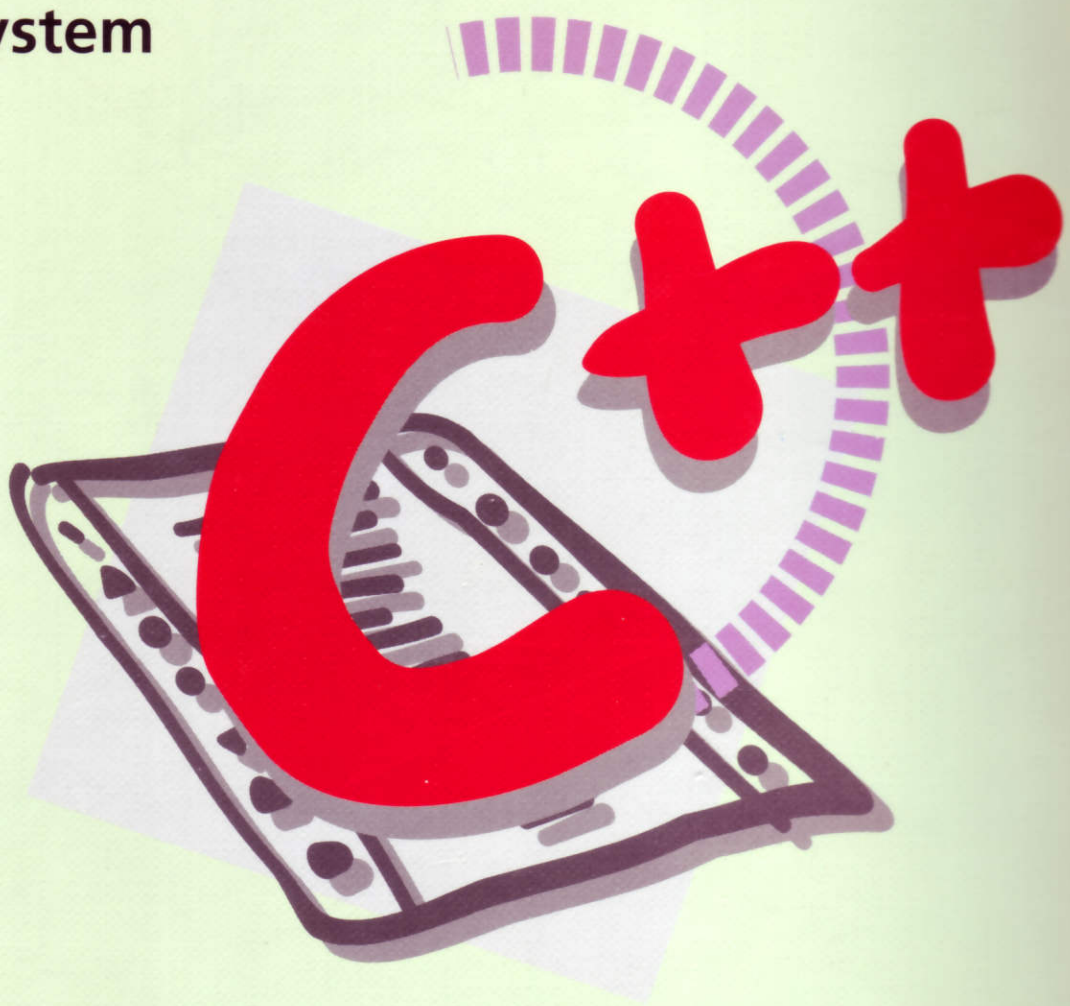
# Maxon C++ 3

## Integriertes Compilersystem




AMIGA®

MAXON computer

*AMIGA*

# MaxonC++

Referenz

MAXON
computer

7.3.5 Generierung von Templatefunktionen ..... 503
7.4 Beispielprogramm: Schon wieder verkettete Listen ..... 504

8. Ausnahmebehandlung ..... 509
8.1 Worum gehts? ..... 509
8.2 Vom Werfen und Auffangen ..... 510
8.3 Von Ausnahmen und ihren Typen ..... 512
8.4 Vom Fangen und wieder Wegwerfen ..... 515
8.5 Von Konstruktoren und Destruktoren ..... 517
8.6 Von erwarteten und unerwarteten Ausnahmen ..... 520
8.7 Beispielprogramm: Das Übliche ..... 523

**Referenz** ..... 525

**1. Die ANSI C Includedateien** ..... 527
1.1 Verschiedene nützliche Funktionen: <stdlib.h> ..... 527
**1.1.1 Umwandlung zwischen Zahlen und Strings** ..... 527
**1.1.2 Mathematische Funktionen** ..... 529
**1.1.3 Zufallszahlen** ..... 531
**1.1.4 Speicherverwaltung, Typen und Zeiger** ..... 532
**1.1.5 Programmende** ..... 534
**1.1.6 Kommunikation mit dem Betriebssystem** ..... 535
**1.1.7 Sortieren und Suchen** ..... 535
1.2 Ein- und Ausgaben: <stdio.h> ..... 536
**1.2.1 Typen, Objekte und Konstanten** ..... 536
**1.2.2 Öffnen und Schließen von Dateien** ..... 538
**1.2.3 Strings und Zeichenketten** ..... 540
**1.2.4 Formatierte Ausgabe** ..... 542
**1.2.5 Formatierte Eingabe** ..... 546
**1.2.6 Dateioperationen** ..... 549
*1.2.6.1 Dateiende und Fehlerbehandlung* ..... 549
*1.2.6.2 Puffern* ..... 550
*1.2.6.3 Externe Dateien* ..... 552
*1.2.6.4 Binäre Daten* ..... 553
*1.2.6.5 Positionieren in Dateien* ..... 554
**1.2.7 Sonstige Funktionen** ..... 555

1.3 Fehlersuche: <assert.h> ....556
1.4 Zeichenklassen: <ctype.h> ....557
1.5 Fehlernummern: <errno.h> ....559
1.6 Ganzzahlige Grenzwerte: <limits.h> ....561
1.7 Mathematische Funktionen: <math.h> ....562
**1.7.1 Umwandlung von Zahlen in Zeichenketten** ....562
**1.7.2 Fließkomma-Berechnungen** ....564
1.8 Haarsträubende Sprünge: <setjmp.h> ....567
1.9 Signale und Ereignisse: <signal.h> ....570
1.10 Variable Argumentlisten: <stdarg.h> ....572
1.11 Implementationsabhängige Definitionen: <stddef.h> ....575
1.12 Zeichenketten und Speicherverwaltung: <string.h> ....576
1.13 Datum und Uhrzeit: <time.h> ....582

**2. Bibliotheken für C++** ....587
2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h> ....587
**2.1.1 Ausgabe** ....587
*2.1.1.1 „ostream" und der Operator „«"* ....587
*2.1.1.2 Basen und Manipulatoren* ....589
*2.1.1.3 Formatierte Ausgabe* ....591
*2.1.1.4 Ausgabe binärer Daten* ....592
**2.1.2 Eingabe** ....593
*2.1.2.1 Texteingaben mit „»"* ....593
*2.1.2.2 Eingabefunktionen auf „istream"* ....594
**2.1.3 Weitere Funktionen** ....596
*2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem* ....596
*2.1.3.2 Fehlererkennung* ....596
2.2 Ströme und Dateien: <fstream.h> ....597
**2.2.1 Klassen für Dateien** ....597
**2.2.2 Dateien öffnen und schließen** ....598
**2.2.3 Fehlererkennung** ....599
**2.2.4 Konstruktoren und Destruktoren** ....600
**2.2.5 Puffer** ....601
**2.2.6 Positionieren in Strömen** ....601

2.3 Ein paar wichtige Definitionen in <new.h> ..... 602
2.4 Ausnahmebehandlung: <exception.h> ..... 605
2.5 Eine String-Library: <tools/str.h> ..... 606
2.5.1 Die Klasse „String“ ..... 606
2.5.2 Member-Funktionen ..... 607
2.5.3 Der Operator „+“ auf Strings ..... 609
2.5.4 Vergleiche ..... 610
2.5.5 Ein- und Ausgabe ..... 611
2.6 Interna von MaxonC++: <streamdefs.h> ..... 612
2.6 Funktionen für originelle Linker-Opt.: <linkerfunc.h> ..... 615

Easy-Objects ..... 617
Einführung ..... 619
Die Ausnahmebehandlung ..... 623
Container und Cursor ..... 623
Das Handlerkonzept ..... 625
Die Fensterklasse WindowC ..... 627
Die Fensterklasse GTWindowC ..... 630
Gadgets und Gadgetlisten ..... 630
Der Layouter ..... 632
Die Regeln zur Bescheibung eines Layouts ..... 633
Der Layoutvorgang ..... 639
Eine Knopfzeile ..... 639
Eine Gadgetspalte ..... 641
Die Grenzen des Layouters ..... 643
Zur Gestaltung der Oberfläche ..... 643
Die Fensterklasse LayouterWindowC ..... 644
Die Fensterklasse StandardWindowC ..... 645
Datei- und Verzeichnissperren ..... 646
Die Dateiklassen ..... 647

# 1. Die ANSI C Includedateien

In diesem Abschnitt werden die Includedateien besprochen, die Bestandteil des ANSI C Standards sind. MaxonC++ deklariert darin aber auch einige zusätzliche eigene Funktionen und Symbole. Diese sind jeweils mit einem **„[M]"** gekennzeichnet.

## 1.1 Verschiedene nützliche Funktionen: <stdlib.h>

In der Include-Datei *<stdlib.h>* werden einige nützliche Funktionen für die unterschiedlichsten Zwecke, z. B. zur Umwandlung von Zahlen und für die Speicherverwaltung, definiert.

### *1.1.1 Umwandlung zwischen Zahlen und Strings*

#### strtol

```
long int strtol(const char *string, char **end, int base)
```

Die Funktion `"strtol"` wandelt eine Zeichenkette in eine ganze Zahl um. Am Anfang von `"string"` darf Leerraum stehen und hinter der Zahldarstellung dürfen noch andere Zeichen folgen. Wird für `"end"` ein Wert ungleich Null angegeben, so wird ein Zeiger auf den nicht umgewandelten Teil der Zeichenkette (z. B. auf die Zeichen, die der Zahldarstellung folgen) in der Variablen, auf die `"end"` verweist, abgelegt. Der Parameter `"base"` gibt die gewünschte Basis an: bei einem Wert zwischen 2 und 36 wird die Zeichenkette mit dieser Basis umgewandelt; ist der Wert 0, hängt die Basis wie in C üblich vom Präfix der Zeichenfolge ab (Hex bei `"0x"` oder `"0X"`, oktal bei `"0"` und andernfalls dezimal).

Wenn das Ergebnis den Bereich von `"long int"` über- oder unterschreitet, wird `"LONG_MAX"` bzw. `"LONG_MIN"` zurückgegeben und die Variable `"errno"` erhält den Wert `"ERANGE"` (siehe *<errno.h>*).

#### strtoul

```
unsigned long strtoul(const char *string, char **end, int base)
```

Diese Funktion ist analog zu `"strtol"`, aber der String wird in ein vorzeichenloses Langwort umgewandelt und im Falle eines Fehlers wird die Konstante `"ULONG_MAX"` zurückgegeben.

## strtod

```
double strtod(const char *string, char **end)
```

Auch diese Funktion wandelt den Anfang einer Zeichenkette in eine Zahl um und zwar in einen "double"-Wert. Wie bei "strtol" wird ein Zeiger auf den nicht umgewandelten Teil der Zeichenkette in "*end" abgelegt, sofern "end" nicht Null ist. Bei einem Überlauf ist das Ergebnis "HUGE_VAL", bei einem Unterlauf wird "0" zurückgegeben.

## strtovl [M]

```
long long strtovl(const char *string, char **end, int base)
```

Diese Funktion entspricht "strtol", wandelt aber in einen 64 Bit breiten "long long int"-Wert um, und im Falle eines Überlaufs ist das Ergebnis "LONGLONG_MAX" bzw. "LONGLONG_MIN".

## strtouvl [M]

```
unsigned long long strtouvl(const char *s, char **end, int base)
```

Auch "strtouvl" führt in gewohnter Weise die Umwandlung eines Strings durch und zwar nach "unsigned long long int". Beim Überlauf wird "ULONGLONG_MAX" als Ergebnis geliefert, sonst ist alles wie bei "strtol".

## atoi

```
int atoi(const char *string)
```

"atoi" wandelt eine dezimale Ziffernfolge in den entsprechenden numerischen Wert um und entspricht damit "(int) strtol(string, 0, 10)".

## atol

```
long atol(const char *string)
```

Auch diese Funktion ist nur eine vereinfachte Form von "strtol" und entspricht "strtol(string, 0, 10)".

### atof

```
double atof(const char *string)
```

Noch eine Kurzform, diesmal aber im Fileßkommabereich: Ein Aufruf von `"atof"` ist äquivalent zu `"strtod(string, 0)"`.

### ERANGE

```
#define ERANGE 1000
```

Diesen Wert erhält die Variable `"errno"`, wenn bei `"strtol"` oder den verwandten Funktionen ein Überlauf auftritt.

### HUGE_VAL

```
#define HUGE_VAL 1.797693134862316E+308
```

Rückgabewert der Funktion `"strtod"` im Falle eines Überlaufs.

## 1.1.2 Mathematische Funktionen

### abs

```
int abs(int i)
```

`"abs"` liefert den absoluten Betrag seines Arguments, also so etwas wie `"(i >= 0? +i : -i)"`.

### labs

```
long int labs(long int l)
```

Genau wie `"abs"`, nur eben für `"long int"`-Werte. Bekanntlich sind in MaxonC++ die internen Darstellungen von `"int"` und `"long"` identisch (jeweils 32 Bit), so daß auch diese beiden Funktionen sich praktisch nicht unterscheiden.

## vlabs [M]

```
long long int vlabs(long long int vl)
```

Noch eine Betragsfunktion, diesmal für `"long long int"` und damit natürlich eine Spezialität von MaxonC++. Ganz schön nervig, daß man in ANSI C nicht überladen kann...

## div

```
div_t div(int nenner, int zaehler)
```

Da bei den üblichen Divisionsalgorithmen gleichzeitig Quotient und Rest ermittelt werden, ist es natürlich Zeitverschwendung, diese beiden Werte in zwei getrennten Rechenschritten zu berechnen. Der Datentyp `"div_t"` ist eine Struktur mit den Einträgen `"quot"` für den Quotienten und `"rem"` für den Rest.

## ldiv_t

```
ldiv_t ldiv(long nenner, long zaehler)
```

Das Ganze noch einmal für `"long"`-Zahlen.

## div_t

```
typedef struct {
     int quot;
     int rem;
     } div_t
```

In Strukturen dieses Typs werden die Ergebnisse der Funktion `"div"` zurückgegeben.

## ldiv_t

```
typedef struct {
     long quot;
     long rem;
     } ldiv_t;
```

ist die Ergebnis-Struktur der Funktion `"ldiv"`.

## 1.1.3 Zufallszahlen

### RAND_MAX

```
#define RAND_MAX 0x7fff
```

Diese Konstante stellt das größtmögliche Ergebnis von `"rand"` dar.

### rand

```
int rand( void )
```

`"rand"` liefert eine Pseudo-Zufallszahl im Bereich von 0 bis RAND_MAX. MaxonC$^{++}$ verwendet hier das additive Kongruenzverfahren, so daß ein Startwert definiert werden muß und zwar mit der folgenden Funktion:

### srand

```
void srand(unsigned int seed)
```

Da die wenigsten Amigas einen Hardware-Zufallszahlengenerator eingebaut haben (am besten solche Dinger, die den Zerfall von Isotopen benutzen, denn solche Vorgänge sind wirklich zufällig), berechnet `"rand"` natürlich nur Pseudo-Zufallszahlen nach einem bestimmten Algorithmus (s. o.). Dieser Algorithmus benötigt einen Startwert, aus dem die erste Zufallszahl berechnet wird (und dann jede weitere Zahl aus der vorhergehenden). `"srand"` setzt diesen Startwert und sollte deshalb vor der ersten Benutzung von `"rand"` aufgerufen werden.

Wenn Sie dabei einen konstanten Wert verwenden, etwa `"srand(4711)"`, erhalten Sie natürlich auch immer dieselbe Folge von „Zufallszahlen". Deshalb ist es üblich, hier einen Wert zu wählen, der von der Systemzeit abhängt, z. B. `"srand(time(0))"`.

### Random [M]

```
unsigned Random(unsigned u)
double Random()
```

Dieser überladene Funktionsname ist eine Spezialität von MaxonC$^{++}$ und kann auch nur benutz werden, wenn *<stdlib.h>* im C$^{++}$-Modus compiliert wird. Mit einen ganzzahligen Argument `"u"` liefert `"Random"` eine Zufallszahl im Bereich von 0 bis u-1, ohne Argument eine Fließkommazahl zwischen 0 und 1 (wobei 0 möglicherweise vorkommen kann, 1 aber nicht). Diese Funk-

tionen benutzen ein ähnliches Verfahren wie `"rand"`, beziehen aber die aktuelle Rasterzeile des Videoausgangs in die Berechnung mit ein. Dadurch muß die Zufallszahlenfolge nicht initialisiert werden und ist deshalb wesentlich „zufälliger" als `"rand"`.

## 1.1.4 Speicherverwaltung, Typen und Zeiger

### malloc

```
void malloc(size_t size)
```

`"malloc"` reserviert `"size"` Bytes Speicher und gibt einen Zeiger darauf zurück oder 0, wenn kein Speicher angefordert werden konnte. `"size_t"` ist ein ganzzahliger Datentyp, der unter MaxonC++ als `"unsigned"` definiert ist und „zufällig" auch der Typ des Ergebnisses von „sizeof" ist.

Mit der Funktion `"set_new_handler"` (siehe *<new.h>*) kann eine Funktion spezifiziert werden, die so lange immer wieder aufgerufen wird, bis genügend Speicher frei ist - oder die Funktion mit `"exit"` o. Ä. aussteigt.

### calloc

```
void *calloc(size_t anzahl, size_t size)
```

Diese Funktion entspricht `"malloc"`, reserviert aber Speicher für einen „anzahl" Elemente großen Vektor des Elementtyps `"size"`.

### free

```
void free(void *ptr)
```

`"free"` gibt einen mit `"malloc"`, `"calloc"` oder `"realloc"` reservierten Speicherbereich wieder frei.

## realloc

```
void *realloc(void *ptr, size_t size)
```

Der Speicherplatzbedarf eines Objekts, auf das `"ptr"` zeigt, wird auf `"size"` geändert und ein Zeiger auf das (möglicherweise verschobene) Objekt zurückgegeben. Natürlich muß es sich um ein Objekt handeln, das mit `"malloc"`, `"calloc"` oder einem anderen `"realloc"`-Aufruf erzeugt wurde.

MaxonC$^{++}$ verfährt dabei wie folgt:

- Ist die neue Größe mit der alten identisch, muß nichts getan werden, und der Zeiger `"ptr"` wird unverändert zurückgegeben.
- Andernfalls wird versucht, einen neuen Speicherbereich der gewünschten Größe zu allozieren, und die passende Anzahl von Bytes umkopiert. Das alte Objekt wird gelöscht und ein Zeiger auf das neue zurückgegeben.
- Kann kein neues Objekt eingerichtet werden, ist das Ergebnis 0, und das ursprüngliche Objekt, auf das `"ptr"` zeigt, bleibt unverändert.

In der ersten Version wurde noch bei Verkleinerung ein Teil des ursprünglichen Objekts freigegeben bzw. bei Vergrößerung versucht, den unmittelbar angrenzenden Speicher anzufordern. Das war zwar effektiv, widerspricht aber den Amiga-Programmierrichtlinien und bereitet insbesondere Tools wie „*MemWall*“ Probleme.

## size_t

```
typedef unsigned size_t
```

`"size_t"` ist der Datentyp, der in C und C$^{++}$ überall da verwendet wird, wo es um die Größe irgendwelcher Objekte geht.

## NULL

```
#define NULL 0
```

Eine Konstante, die manche Leute gern für Zeiger des entsprechenden Werts verwenden. Eine Definition `"((void*)0)"` wie in vielen anderen C-Systemen ist hier übrigens nicht möglich, da C$^{++}$ keine direkte Konvertierung von `"void*"` in andere Zeigertypen kennt.

## wchar_t

```
typedef int wchar_t
```

Der Datentyp von „langen" Zeichenkonstanten, z. B.

```
wchar_t C = L'x';
```

### 1.1.5 Programmende

## exit

```
void exit(int res)
```

Diese Funktion beendet das Programm und entspricht einem `"return res"` aus der Funktion `"int main()"`. Mit `"atexit"` eingehängte Funktione werden ausgeführt, Destruktoren für globale Variablen aufgerufen und Resourcen (Dateien, Speicher usw.) freigegeben. Das Ergebnis Null zeigt normalerweise an, daß das Programm erfolgreich ausgeführt wurde, und jeder andere Wert wird als Fehlernummer interpretiert.

## atexit

```
int atexit(void (*funktion)(void))
```

teilt der Laufzeitbibliothek mit, daß die angegebene parameterlose Funktion am Programmende ausgeführt werden soll.

## abort

```
void abort(void)
```

Das Programm wird gnadenlos abgebrochen, ganz ohne Destruktoren, `"atexit"`-Funktionen und Resourcen-Freigabe.

## 1.1.6 Kommunikation mit dem Betriebssystem

### getenv

```
char *getenv(const char *name)
```

Die Funktion liefert den Wert der Environment-Variablen `"name"` und liefert einen Zeiger darauf oder Null, wenn keine derartige Variable existiert.

Diese Funktion benötigt mindestens die Kickstart-Version 2.0 (37.175) und liefert bei früheren Versionen stets Null.

### system

```
int system(const char *command)
```

Die Funktion führt den in `"command"` beschriebenen CLI-Befehl aus und liefert dessen Rückgabewert als Ergebnis. Auch diese Funktion benötigt mindestens Betriebssystemversion 2.0 und liefert andernfalls immer `"-1"`. Die Ausgaben der Funktion erfolgen über die Standardausgabe des Prozesses - und in der Entwicklungsumgebung MCPP ist dies bei Betriebssystemversion 1.2 oder 1.3 leider die Umgebung, aus der MaxonC++ einmal gestartet wurde, nämlich das entsprechende CLI-Fenster oder `"Nil:"` beim Start von der Workbench. Ab Betriebssystemversion 2.0 werden die Ausgaben korrekt umgelenkt.

## 1.1.7 Sortieren und Suchen

### qsort

```
void qsort(void *vektor, size_t anzahl, size_t size,
int (*compare)(const void *, const void *))
```

Diese Funktion sortiert den Vektor `"vektor[0] ... vektor[anzahl-1]"`, dessen Elemente jeweils die Größe `"size"` haben, mit der angegebenen Vergleichsfunktion. In Abschnitt 2.3.4 des Tutorials wird `"qsort"` ausführlich beschrieben.

## bsearch

```
void *bsearch(const void *obj, const void *vektor,
  size_t anzahl, size_t size,
  int (*compare)(const void *, const void *))
```

sucht im Vektor `"vektor[0] ... vektor[anzahl-1]"`, dessen Elemente jeweils die Größe `"size"` haben, anhand der angegebenen Vergleichsfunktion ein Element, das mit `"*obj"` übereinstimmt. Da die Suche binär erfolgt, muß der Vektor dafür (z. B. mit `"qsort"`) sortiert sein. Auch diese Funktion wird in Abschnitt 2.3.4 des Tutorials näher erläutert.

# 1.2 Ein- und Ausgaben: <stdio.h>

## *1.2.1 Typen, Objekte und Konstanten*

## FILE

```
typedef struct stream FILE
```

Der Datenstrom ist das Hauptkonzept der Ein- und Ausgabe in ANSI C und entspricht einer Datei, die aber nicht unbedingt wie eine Datei aussehen (also auf einer Diskette oder Festplatte liegen) muß, sondern auch z. B. ein Terminal, ein Window oder ein Drucker sein kann. Der Datentyp, der solche Strom-Objekte beschreibt, heißt `"struct stream"` oder `"FILE"`.

## std__in, std__out, std__err [M]

```
extern FILE std__in, std__out, std__err
```

Dies sind die Namen der drei Standard-Dateien: `"std__in"` für Eingaben des Programms, `"std__out"` für Ausgaben und `"std__err"` für Fehlermeldungen. Auf dem Amiga sind der Ausgabe- und der Fehlerkanal (leider) identisch. Beim Programmstart sind diese Dateien geöffnet und initialisiert (auch wenn sie beim Workbench-Start ins Nirvana zeigen) und müssen am Programmende auch nicht geschlossen werden.

Die drei Datenströme müssen keineswegs in jedem C-System diese Namen tragen, sondern können durchaus anders heißen. Deshalb sollte man sie nie unter diesen Bezeichnern ansprechen, sondern nur mit den drei folgenden Makros:

## stdin, stdout, stderr

```
#define stdin (&std__in)
#define stdout (&std__out)
#define stderr (&std__err)
```

Die meisten Funktionen, die etwas mit Ein- und Ausgabe von Daten zu tun haben, erwarten als Argument keinen Datenstrom, sondern einen Zeiger auf einen solchen. Deshalb gibt es diese drei Makros, die jeweils einen Zeiger auf eine Standard-Datei liefern.

## NULL

```
#define NULL 0
```

Die allseits beliebte Konstante (für Zeiger) wird auch in dieser Include-Datei definiert.

## size_t

```
typedef unsigned size_t
```

`"size_t"` ist der Standardname für den Typ, den Ergebnisse von `"sizeof"` haben.

## EOF

```
#define EOF (-1)
```

Die Konstante `"EOF"` wird an verschiedenen Stellen zum Anzeigen des Dateiendes bzw. eines Fehlers benutzt.

## 1.2.2 Öffnen und Schließen von Dateien

### fopen

```
FILE *fopen(const char *name, const char *modus)
```

öffnet die Datei des angegebenen Namens und mit dem angegebenen Modus, erzeugt dazu eine FILE-Struktur und gibt einen Zeiger darauf zurück. Kann die Datei nicht geöffnet werden, liefert die Fuktion Null.

Es gibt die folgenden Modi:

`"r"` Vorhandene Datei zum Lesen öffnen.

`"w"` Datei zum Schreiben öffnen bzw. erzeugen und dabei evtl. vorhandene Datei überschreiben.

`"a"` Existierende Datei zum Anhängen ans Dateiende öffnen bzw. neue Datei erzeugen und zum Schreiben öffnen.

`"r+"` Vorhandene Datei zum Lesen und Schreiben öffnen oder ggf. eine neue Datei erzeugen.

`"w+"` Eine neue Datei erzeugen (ggf. existierende Datei löschen) und zum Lesen und Schreiben öffnen.

`"a+"` Vorhandene Datei fürs Lesen und Schreiben öffnen (ggf. neue Datei anlegen), wobei Schreib-/Leseposition auf Dateiende gesetzt wird.

ANSI C unterscheidet zwischen Text- und Binärdateien. Die oben aufgeführten Modi öffnen eine Datei als Textdatei. Will man eine Binärdatei öffnen, ist ein `"b"` an die zweite oder dritte Stelle des Modus-Strings zu setzen, z. B. `"rb"`, `"rb+"` oder `"r+b"`.

Auf dem Amiga (wie auch unter UNIX) sind die beiden Dateimodi identisch, und so ist es egal, ob Sie das zusätzliche `"b"` angeben oder nicht. MS-DOS Computerund die MessyDOS-Kopie Atari ST dagegen müssen tricksen und bei Textdateien dem Programm vorgaukeln, daß die Folge `"Linefeed+Return"` aus nur einem einzigen Zeichen besteht.

### fclose

```
int fclose(FILE *f)
```

Eine geöffnete Datei muß auch wieder geschlossen werden. Die Funktion `"fclose"` schließt eine Datei, die mit `"fopen"` geöffnet wurde, und gibt die zugehörige `"FILE"`-Struktur frei. Vorher werden natürlich alle Puffer geleert (wie bei `"fflush"`). Das Ergebnis ist Null oder

`"EOF"` bei einem Fehler, aber auf dem Amiga kann beim Schließen einer Datei nichts schiefgehen.

## freopen

```
FILE *freopen(const char *name, const char *modus, FILE *file)
```

leitet einen bereits geöffneten Datenstrom in eine andere Datei um. Die physikalische Datei, mit der `"file"` bisher verbunden war, wird geschlossen, eine neue Datei mit dem angegebenen Namen und Modus geöffnet und ein Verweis darauf in die vorhandene FILE-Struktur gepackt.

Diese Funktion dient vor allem dazu, die Standard-Datenströme `"stdin"`, `"stdout"` oder `"stderr"` in eine andere physikalische Datei umzuleiten.

## FILENAME_MAX

```
#define FILENAME_MAX 200
```

Der Amiga kennt es bekanntlich so gut wie keine Beschränkung der Länge von Dateinamen, aber genau diese maximale Länge soll `"FILENAME_MAX"` eigentlich angeben. Die Konstante wurde mehr oder weniger willkürlich mit 200 definiert, denn länger wird ein Dateiname einschließlich Pfad wohl kaum werden.

## FOPEN_MAX

```
#define FOPEN_MAX 99999
```

Dank dynamischer Datenorganisation ist die Anzahl der Dateien, die Sie mit MaxonC$^{++}$ öffnen dürfen, nicht begrenzt und die Konstante `"FOPEN_MAX"` deshalb ziemlich Banane. Begrenzt ist diese Anzahl lediglich dadurch, daß irgendwann der Speicher voll ist oder das Betriebssystem sich sonstwie beschwert.

## 1.2.3 Strings und Zeichenketten

### fgetc

```
int fgetc(FILE *f)
```

liest ein Zeichen aus der Datei `"f"` und wandelt es zuerst in `"unsigned char"` und dann in `"int"` um, denn bekanntlich gibt es in C keine klare Trennung zwischen Zahlen und Zeichen. Wenn das Dateiende überschritten wurde oder sonst ein Fehler auftrat, wird `"EOF"` geliefert.

### <$igetc

```
int getc(FILE *f)
```

Diese Funktion ist witzigerweise mit `"fgetc"` identisch, allerdings dürfen Implementationen sie aus nicht nachvollziehbaren Gründen in Form eines Makros definieren. Wie erst im Schwinden der Dominanz des Signifikats die semantische Struktur der Horizontflucht, des steten Aufschubs der Signifikation, entsteht, so wird andererseits mit der Substitution des Signifikats durch den Funktionszusammenhang der Signifikanten die Horizonterfahrung als Motorik des ständigen Verweisens ungültig.

### getchar

```
int getchar (void)
```

ist äquivalent zu `"getc(stdin)"`, liest also ein Zeichen aus der Standard-Eingabe.

### ungetc

```
int ungetc(int c, FILE *f)
```

Manchmal merkt man erst, daß man genug gelesen hat, wenn man zuviel gelesen hat. Die Funktion `"ungetc"` stellt ein bereits gelesenes Zeichen in eine Eingabedatei zurück, so daß es beim nächsten Lesevorgang erneut gelesen wird. Bei einer „echten" Datei kann man sich `"ungetc"` wie das Zurücksetzten des Lesezeigers um ein Zeichen vorstellen, aber bei einer interaktiven Eingabe geht das so natürlich nicht. Deshalb hat jeder Datenstrom einen Puffer, in dem solche „zurückgestellten" Zeichen aufgenommen werden. Es paßt aber immer nur genau ein Zeichen in diesen Puffer.

## fgets

```
char *fgets(char *s, int n, FILE *f)
```

"fgets" liest eine Zeile mit höchstens "n-1" Zeichen aus "f" und legt sie einschließlich eines abschließenden Null-Bytes im String "s" ab. Als Zeilenende gilt entweder ein Linefeed-Zeichen ('\n') oder das Dateiende (EOF). In beiden Fällen wird das jeweilige Zeichen aber nicht in die Stringvariable aufgenommen. Wurde das Dateiende bereits überschritten oder trat ein Fehler auf, gibt "fgets" Null, andernfalls "s" zurück.

## gets

```
char *gets(char *s)
```

entspricht "fgets(s, STREAM_MAXSTRING, stdin)", liest also eine Zeile von der Standardeingabe.

## STREAM_MAXSTRING [M]

```
#define STREAM_MAXSTRING 80
```

Diese Konstante gibt an wieviele Zeichen maximal von "gets" gelesen werden, ist aber nicht Bestandteil des ANSI C-Standards.

## fputc

```
int fputc(int c, FILE *f)
```

Natürlich kann man nicht nur Daten lesen, sondern sie auch schreiben. "fputc" ist die wohl rudimentärste Ausgabefunktion und schreibt ein Zeichen "c" in eine Datei "f". Das Ergebnis ist "EOF" bei Fehler oder andernfalls das ausgegebene Zeichen.

## putc

```
int putc(int c, FILE *f)
```

Ähnlich wie "getc" und "fgetc", ist diese Funktion mit "fputc" identisch, darf aber als Makro definiert sein. Sie wissen ja, erst im Schwinden der Dominanz des Signifikats entsteht...

## putchar

```
int putchar(int c)
```

schreibt das Zeichen in die Standard-Ausgabe und ist damit identisch mit: `"putc(c, stdout)"`.

## fputs

```
int fputs(const char *s, FILE *f)
```

Diese Funktion schreibt eine Zeichenfolge, die wie üblich mit einen Null-Zeichen enden muß, in die Datei `"f"`. Anstelle des abschließendne Nullbytes wird ein Zeilenvorschub, also ein `"\n"`, geschrieben.

## puts

```
int puts(const char *s)
```

dient ebenfalls zur Ausgabe von Zeichenketten und ist äquivalent zu `"fputs(s, stdout)"`.

### 1.2.4 Formatierte Ausgabe

## printf

```
int printf(const char *format, ...)
```

Diese ungemein nützliche Funktion ist die Standard-Ausgabefunktion in C, jedenfalls für Textdateien. Im Prinzip wird dabei die Zeichenkette `"format"` ausgegeben, nur an bestimmten Stellen werden andere Daten eingesetzt. Diese „bestimmten Stellen" sind Umwandlungsanweisungen, die stets mit einem `"%"` anfangen, mit einem Buchstaben enden und dazwischen noch zusätzliche Argumente haben dürfen. Ein Beispiel:

```
int a=17, b = 4;
printf("Die Summe von %d und %d ist %d.\n", a, b, a+b);
```

Die Zeichenfolge `"%d"` weist `"printf"` an, aus den sonstigen Argumenten das nächste zu nehmen, als „int" zu interpretieren und dezimal auszugeben. Also erzeugt der obige Funktionsaufruf folgende Ausgabe:

**Die Summe von 17 und 4 ist 21.**

Natürlich gibt es nicht nur `"%d"`, sondern eine ganze Fülle von Format-Anweisungen, die folgendes in der genannten Reihenfolge enthalten können oder müssen:

- Das erste Zeichen ist, wie bereits erwähnt, stets `"%"`.
- Eine optionale Folge von Flags in beliebiger Reihenfolge: Ein `"-"` bewirkt eine linksbündige Ausgabe der Daten (nur sinnvoll, wenn eine Feldbreite angegeben wird, siehe unten), `"+"` sorgt dafür, daß eine Zahl auf jeden Fall mit Vorzeichen (also ggf. `"+"`) ausgegeben wird, wogegen ein Leerzeichen veranlaßt, daß bei vorzeichenlosen Zahlen ein Leerzeichen an den Anfang gesetzt wird. Das Flag `"0"` füllt bei Zahlen-Ausgaben das Ausgabefeld mit Nullen statt mit Leerzeichen auf (ebenfalls nur sinnvoll, wenn Feldbreite gesetzt), und `"#"` hat bei einigen Ausgabeformaten eine besondere Wirkung: Hexadezimalen Ausgaben wird ein `"0x"`, oktalen eine `"0"` vorangestellt, Fließkommaausgaben enthalten auf jeden Fall einen Dezimalpunkt und bei den Anweisungen `"g"` und `"G"` werden Nullen am Ende nicht unterdrückt. Welch' ein Satz!
- Die Feldbreite und zwar als dezimale Ziffernfolge. Die Ausgabe des umgewandelten Arguments erfolgt (sofern nicht anders spezifiziert) linksbündig in einem Feld, das mindestens diese Breite hat, wobei bei Bedarf mit Leerzeichen (oder Nullen, siehe oben) aufgefüllt wird.
- Die Genauigkeit, die mit einem `"."` anfängt (damit `"printf"` sie von der Feldbreite unterscheiden kann), ebenfalls als dezimale Ziffernfolge. Bei Zeichenketten legt dies die maximale Anzahl von Zeichen fest, bei ganzzahligen Ausgaben die minimale Anzahl von Ziffern (nach links wird dann mit Nullen aufgefüllt) und bei Fließkommazahlen die Anzahl der Dezimalstellen.
- Ein Buchstabe als Längenangabe: Bei `"h"` ist das Argument als `"short"` oder `"unsigned short"` zu betrachten, `"l"` weist es als `"long int"` aus und bei `"L"` ist es entweder `"long double"` oder `"long long"` oder `"unsigned long long"`.
- Das Umwandlungszeichen, z. B. `"d"` für ganzzahlige dezimale Ausgaben.

Außer dem `"%"` am Anfang und dem Umwandlungszeichen am Ende sind alle genannten Argumente optional. Als Genauigkeit oder Feldbreite kann anstelle einer Ziffernfolge `"*"` gesetzt werden. In diesem Fall steht der fehlende numerische Wert als nächstes `"int"`-Argument in der Argumentliste.

Es gibt folgende Umwandlungszeichen:

| Zeichen | Argumenttyp | Art der Ausgabe |
|---|---|---|
| `d` | int | Dezimal |
| `i` | int | Wie `"d"` |
| `o` | int, unsigned | Vorzeichenlos oktal, ohne `"0"` am Anfang |
| `x` | int, unsigned | Vorzeichenlos hexadezimal ohne führendes `"0x"` und mit kleinen Buchstaben |
| `X` | int, unsigned | Wie `"x"`, aber mit Großbuchstaben |
| `u` | unsigned | Dezimal ohne Vorzeichen |
| `c` | int | Einzelnes Zeichen als `"unsigned char"` |
| `s` | char* | String, der mit Nullbyte endet |
| `f` | double | Fließkomma-Ausgabe in der Form `"[-]xxx.yyy"`. Die Genauigkeit legt die Anzahl der Nachkommastellen fest, Default ist 6. |
| `e` | double | Exponentielle Darstellung `"x.yyyyyeõzzz"`, wobei die Genauigkeit wieder die Zahl der Nachkommastellen angibt (Default 6). |
| `E` | double | wie `"e"`, aber mit großen `"E"`: `"x.yyyyyEõzzz"` |
| `g` | double | Bei Argumenten, deren Exponent zwischen der Genauigkeit (z. B. 6 als Default) und -4 liegt, erfolgt die Ausgabe wie bei `"f"`, andernfalls wie bei `"e"`. |
| `G` | double | Entspricht `"g"`, aber es wird zwischen `"f"` und `"E"` gewählt. |
| `p` | void* | Hexadezimale Speicheradresse |
| `n` | int* | Bewirkt keine Ausgabe, sondern legt die Anzahl der bisher ausgegebenen Zeichen in der Variablen, auf die das Argument zeigt, ab. |
| `%` | - | Gibt ein Prozentzeichen aus. |

Das Ergebnis von `"printf"` ist die Anzahl der ausgegebenen Zeichen oder negativ, wenn ein Fehler auftrat. Bitte denken Sie stets daran, daß `"printf"` eine Funktion ohne Netz und doppelten Boden ist und fehlerhafte Argumente bestenfalls zu chaotischen Ausgaben und schlimmstenfalls zu einem Systemabsturz führen können.

## fprintf

```
int fprintf(FILE *f, const char *format, ...)
```

Diese Funktion arbeitet eigentlich genau wie `"printf"`, aber die Daten werden nicht über den Standard-Ausgabestrom, sondern in die Datei `"f"` ausgegeben. Andersrum formuliert, ist `"printf(irgendwas)"` eine Abkürzung für `"fprintf(stdout, irgendwas)"`.

## sprintf

```
int sprintf(char *s, const char *format, ...)
```

Auch diese Funktion entspricht `"printf"`, aber die Ausgaben werden nirgendwo wirklich ausgegeben, sondern im String `"s"` abgelegt und mit einem Nullzeichen abgeschlossen. Sie als Programmierer sind selbst dafür verantwortlich, daß der Ziel-String groß genug ist, denn andernfalls wird (wie in C nicht anders zu erwarten) gnadenlos über den Vektor hinaus geschrieben, und der Guru gerät ins Meditieren.

## vprintf, vfprintf, vsprintf

```
typedef unsigned va_list;

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *f, const char *format, va_list arg);

int vsprintf(char *s, const char *format, va_list arg);
```

Diese Funktionen sind absolut äquivalent zu ihren Entsprechungen ohne das `"v"` am Anfang, aber die Argumente, die sonst dem Formatstring folgen, werden hier durch eine variable Argumentenliste (`"va_list"`) übergeben. Bitte lesen Sie dazu auch das Kapitel über *<stdarg.h>* oder fragen Sie Ihren Arzt oder Apotheker.

## 1.2.5 Formatierte Eingabe

### scanf

```
int scanf(const char *format, ...)
```

Was `"printf"` für die Ausgabe, ist `"scanf"` für die Eingabe. Der Formatstring ist ähnlich aufgebaut wie bei `"printf"`, hat aber eine andere Bedeutung:

Leerzeichen oder Tabulatoren in der Formatzeichenkette werden ignoriert. Bei allen anderen Zeichen außer `"%"` erwartet `"scanf"`, daß dieses Zeichen als nächstes in der Eingabe folgt (eventuell mit vorangehenden Leerzeichen) und liest es bzw. bricht ab, wenn das Zeichen nicht folgt. Wieder leitet `"%"` Umwandlungsanweisungen ein.

Einem `"%"` kann optional ein `"*"` folgen (dann werden zwar Zeichen gelesen und umgewandelt, aber nirgendwo abgelegt), oder eine Zahl, die die maximale Feldbreite festlegt, also die maximale Anzahl von Zeichen, die bei dieser Umwandlung gelesen werden sollen. Die optionalen Zeichen `"h"`, `"l"` und `"L"` legen wie bei `"printf"` die Datenbreite fest.

Zu jeder Umwandlungsanweisung gehört in der restlichen Argumentliste ein Zeiger auf eine Variable, es sei denn, hinter dem `"%"` folgt ein `"*"` (siehe oben). Bei einer Umwandlung überliest `"scanf"` zunächst alle Zwischenraumzeichen einschließlich Zeilentrennern, liest eine Zeichenfolge ein, die der Umwandlungsanweisung entspricht, z. B. eine Ziffernfolge bei dezimalganzzahliger Umwandlung, wandelt diese um und legt das Ergebnis an der beschriebenen Stelle ab. Wenn ein Fehler auftritt, bricht `"scanf"` sofort ab.

Bei den bereits mehrfach erwähnten Umwandlungszeichen handelt es sich um die Folgenden:

| Zeichen | Argument | Erwartete Eingabedaten und Umwandlung |
|---|---|---|
| `d` | int* | Dezimal, ganzzahlig |
| `i` | int* | Ganzzahlig, wahlweise dezimal, oktal (mit `"0"` beginnend) oder hex (mit `"0x"` oder `"0X"` am Anfang) |
| `o` | int* | Oktal, ganzzahlig |
| `x` | int* | Hexadezimal mit oder ohne `"0x"` |
| `c` | char* | So viele Zeichen, wie die Feldbreite angibt (Default ist 1), wobei Leerzeichen, Zeilentrenner o. Ä. nicht überlesen werden. Will man ein Zeichen lesen, vorher aber Zwischenraumzeichen überlesen, ist die Umwandlung `"%1s"` zu wählen. Im Gegensatz zu `"%s"` wird bei `"%c"` kein Nullbyte angehängt. |
| `s` | char* | Überliest zuerst Leerzeichen und Tabulatoren und liest dann eine Folge von Nicht-Zwischenraum-Zeichen. Ein Nullbyte wird an das Ergebnis angehängt. |
| `e,f,g` | float* | Fließkommazahl in nahezu beliebigem Format (mit oder ohne Voerzeichen, mit oder ohne Exponent, der ein großes `"E"` oder ein kleines `"e"` haben darf usw.) |
| `p` | int* | Hexadezimale Zahl |
| `n` | int* | Liest nichts, sondern legt im Argument die Anzahl der bisher gelesenen Zeichen ab. |
| `[...]` | char* | Längste Zeichenkette, die ausschließlich aus den Zeichen zwischen `"["` und `"]"` besteht, z. B. Ziffernfolgen bei Umwandlungsoperator `"%[0123456789]"`. Auch hier kann die Länge durch Setzen einer Feldbreite beschränkt werden. Will man das Zeichen `"]"` aufnehmen, ist es an den Anfang zu setzen (z. B. `"%[ ]abcde]"`). Die gelesenen Zeichen werden wie üblich mit einem `'\0'` abgeschlossen. |
| `[^...]` | char* | Liest die längste Zeichenkette, deren Zeichen sich nicht in der angegebenen Menge befinden. |
| `%` | - | Erwartet ein `"%"`, legt es aber nirgendwo ab. |

Natürlich kann statt eines Zeigers auf `"int"` stets ebenso gut ein Zeiger auf `"unsigned int"` verwendet werden. Bei ganzzahligen Umwandlungsoperationen wird das zugehörige Zeigerargument durch die Längenangabe ‚h' als `"short"` betrachtet, `'l'` macht daraus `"long"`

und `'L'` `"long long"`. Bei den Fließkommaoperationen ist `'l'` für `"double"`-Variablen und `'L'` für `"long double"`-Variablen zu setzen.

Beispiel:

```
int i,j;
short s;
char c[80];

scanf("Test %d %i, %hi %79s", &i, &j, &s, c)
```

akzeptiert z. B. die Eingabe

```
Test 42 0x686b, 99 Rest
```

oder auch

```
Test42 26731      ,99Rest
```

Das Ergebnis von `"scanf"` ist in beiden Fällen 4, und in die Variablen werden die Werte `"42"`, `"26731"`, `"99"` und `"Rest"` eingelesen.

Mehr noch als `"printf"` ist `"scanf"` eine höchst unsichere Funktion, da keine wirkliche Typprüfung der Argumente vorgenommen wird. Wenn man „nur" ein `"&"` vergißt (und damit den Wert einer Variablen statt ihrer Adresse als Argument übergibt), ist der Hangup oft nicht mehr zu vermeiden.

## fscanf

```
int fscanf(FILE *f, const char *format, ...)
```

Dies ist das Äquivalent zu `"scanf"` für beliebige Dateien.

```
scanf(arglist)
```

ist gleichbedeutend mit

```
fscanf(stdin, arglist)
```

wobei `"arglist"` hier nichts mit Niedertracht und Heimtücke zu tun hat, sondern eine Argument-Liste darstellen soll.

### sscanf

```
int sscanf(char *s, const char *format, ...);
```

ist ebenfalls eine andere Form von `"scanf"`. Hier wird die Eingabe aus dem String `"s"` gelesen. Beispielsweise wandelt

```
sscanf(s, "%i", &n)
```

die Ziffernfolge `"s"` in eine Zahl um und leistet dabei ähnliches wie

```
n = atoi(s)
```

## 1.2.6 Dateioperationen

### 1.2.6.1 Dateiende und Fehlerbehandlung

### feof

```
int feof(FILE *f)
```

liefert einen Wert ungleich Null, wenn in der genannten Datei das Dateiende überschritten wurde.

### ferror

```
int ferror(FILE *f)
```

Diese Funktion schaut nach, ob bei der Datei `"f"` bisher ein Fehler aufgetreten ist und vermerkt wurde, und liefert die Fehlernummer oder andernfalls Null.

### clearerr

```
void clearerr(FILE *f)
```

löscht die Dateiende- und Fehlervermerke einer Datei, so daß anschließend `"feof"` und `"ferror"` jeweils Null ergeben.

### 1.2.6.2 Puffern

## setvbuf

```
int setvbuf(FILE *f, char *buf, int mode, unsigned size)
```

Normalerweise gehen Ausgabeoperationen erheblich schneller, wenn man die Daten nicht in kleinen Kleckermengen (z. B. Zeichenweise) schreibt, sondern sie erst einmal in einem Puffer sammelt und dann auf einen Schlag schreibt. Das gilt vor allem unter den alten Amiga-Systemversionen mit diesem unsäglich lahmen BCPL-DOS, aber auch unter 2.0 und höchstwahrscheinlich auch allen folgenden Versionen. Der Grund ist ganz einfach der, daß bei jeder Ausgabeoperation ein erheblicher Overhead nötig ist: Das Programm ruft eine DOS-Funktion auf, das DOS gibt an das File-System weiter, welches wiederum ein Device befragt, und erst das betätigt dann (z. B.) den Controller, der dann mit der Festplatte kommuniziert... Dieser Overhead ist für jede Datengröße praktisch identisch, so daß es natürlich Zeit spart, wenn man das ganze Prozedere nur einmal für einen großen Datenblock durchexerziert und nicht für jedes Byte einzeln. Analog gilt das natürlich auch für Eingaben aus einer Datei.

Die Funktion `"setvbuf"` versieht eine Datei mit einem solchen Puffer. Man kann sie für jede Datei nur einmal aufrufen, und zwar unmittelbar nach dem Öffnen, d. h. vor jeder Ein- und Ausgabe oder sonstigen Operation. `"buf"` zeigt dabei auf einen hinreichend großenSpeicherbereich, beim Wert Null legt `"setvbuf"` selbstständig einen solchen Puffer an und löscht ihn beim Schließen der Datei auch wieder. Der Parameter `"size"` gibt die gewünschte Puffergröße an. Erfahrungsgemäß sind hier einige hundert Bytes genug und eine weitere Steigerung bringt nicht mehr viel, und `"mode"` gibt einen der im folgenden beschriebenen Puffermodi an. Das Ergebnis von `"setvbuf"` ist Null, wenn alles klar ging.

## IOFBF

```
#define _IOFBF 1
```

Beim Modus `"_IOFBF"` puffert `"setvbuf"` vollständig, d. h. der Puffer wird immer komplett gefüllt, bevor eine neue Operation erfolgt.

## IOLBF

```
#define _IOLBF (-1)
```

Bei Ausgaben wird der Puffer im Modus `"_IOLBF"` höchstens bis zum ersten Zeilenende gefüllt. Dieser Modus ist z. B. auch bei Bildschirmausgaben brauchbar und bringen sogar eine gewisse Beschleunigung.

## IONBF

```
#define _IONBF 0
```

Es wird nichts gepuffert. Einen Aufruf von `"setvbuf"` mit diesem Modus kann man sich also eigentlich sparen.

## setbuf

```
void setbuf(FILE *f, char *buf)
```

Dies ist eine Spar-Version von `"setvbuf"`, bei der `"buf"` auf einen Puffer der Größe `"BUFSIZ"` zeigen. `"setbuf"` legt nicht selbstständig einen Speicherbereich der geforderten Größe an.

```
char b[BUFSIZ];
FILE *f;

setbuf(f, b);
```

ist äquivalent zu

```
setvbuf(f, b, _IOFBF, BUFSIZ);
```

Fehler können bei `"setbuf"` nicht auftreten, und somit ist das Ergebnis `"void"`.

## BUFSIZ

```
#define BUFSIZ 200
```

ist die Puffergröße, die bei `"setbuf"` angenommen wird.

## fflush

```
int fflush(FILE *f)
```

klingt zwar irgendwie gestottert, wirkt aber in Wirklichkeit einer stoßweisen Stotter-Ausgabe (will sagen: Schreiben mit Puffer) entgegen: Die Daten, die sich gerade im Puffer der Datei befinden, werden geschrieben, und der Puffer wird gelöscht. Bei Eingabe-Dateien hat `"fflush"` keine Wirkung, und wenn als Argument Null gewählt wird, werden alle gerade geöffneten Dateien geflusht.

### 1.2.6.3 Externe Dateien

## remove

```
int remove(const char *name)
```

entfernt die Datei dieses Namens, entspricht also dem CLI-Befehl `"delete"`. Hat das geklappt, ist das Resultet Null, andernfalls die Fehlernummer.

## rename

```
int rename(const char *oldname, const char *newname)
```

benennt die Datei `"oldname"` in `"newname"` um, oder versucht es zumindest. Im Erfolgsfall wird Null als Ergebnis geliefert und andernfalls eine Fehlernummer.

## tmpnam

```
char *tmpnam (char s[L_tmpnam])
```

Manchmal steht man vor der Situation, daß man einige Daten in temporären Dateien ablegen will. Die Funktion `"tmpnam"` liefert dafür Dateinamen, die garantiert mit keiner existierenden Datei übereinstimmen: Sie beginnen mit `"t:TMP_"`, und im Rest der Zeichenfolge sind die Adresse der Taskstruktur, die laufende Nummer des `"tmpnam"`-Aufrufs, das Datum und die Uhrzeit kodiert.

Der Parameter `"s"` muß auf eine Stringvariable der Länge (mindestens) `"L_tmpnam"` zeigen, in der das Ergebnis abgelegt wird, oder Null sein, dann wird der Dateiname in einem internen Buffer abgelegt. In beiden Fällen wird als Ergebnis ein Zeiger auf den String geliefert. `"tmpnam"` erzeugt keine Datei, sondern schlägt nur einen Namen vor. Außerdem stehen höchstens `"TMP_MAX"` verschiedene Namen zur Verfügung, aber bei MaxonC++ sind das mehr als genug.

`L_tmpnam`

```
#define L_tmpnam 40
```

Die Konstante gibt an, wie groß ein String wenigstens sein muß, um das Ergebnis von `"tmpnam"` aufzunehmen.

`TMP_MAX`

```
#define TMP_MAX 0x10000
```

"tmpnam" kann TMP_MAX mal aufgerufen werden, bevor sie sich wiederholt. In MaxonC++ sind das offensichtlich mehr als genug mögliche Dateinamen, aber darauf sollte man sich nicht verlassen - man denke hier nur an MesSy-DOS, wo Dateinamen nur 8+3 Zeichen lang sein dürfen. Übrigens muß man sich ganz schön beeilen, wenn man mit "tmpnam" wirklich zweimal denselben Namen erhalten will, denn MaxonC++ codiert ja auch die Uhrzeit in den Namen ein.

## tmpfile

```
FILE *tmpfile (void)
```

ist so etwas wie die Kombination von "tmpnam" mit "fopen" und einer Löschautomatik: Eine temporäre Datei wird mit dem Modus "wb+" geöffnet, und beim Schließen dieser Datei oder am Programmende wird sie automatisch wieder gelöscht. Wenn das Ganze ein Schlag ins Wasser war, liefert "tmpfile" Null, andernfalls einen Zeiger auf die Datei.

### 1.2.6.4 Binäre Daten

## fread

```
unsigned fread(void *ptr,
unsigned size, unsigned n, FILE *f)
```

"fread" liest aus der Datei "f" in den Vektor "ptr" maximal "n" Objekte der Größe "size". Bitte fragen Sie nicht, warum man hier und bei "fwrite" erst "size" und dann "n" angeben soll, während es bei "calloc", "qsort" und "bsearch" genau umgekehrt ist. "fread" liefert als Ergebnis die Anzahl der gelesenen Objekte (nicht Bytes!), die natürlich keineswegs größer, aber durchaus geringer als "n" sein kann (z. B. wenn das Dateiende erreicht wurde).

## fwrite

```
unsigned fwrite(const void *ptr, unsigned size, unsigned n, FILE *f)
```

ist das Gegenstück zu "fread" und schreibt "n" Elemente der Größe "size" aus dem Vektor "ptr" in die Datei "f". Resultat ist die Anzahl der geschriebenen Objekte, die im Falle eines Fehlers kleiner als "n" ist.

### 1.2.6.5 Positionieren in Dateien

## fseek

```
int fseek(FILE *f, long offset, int modus)
```

setzt den Schreib- bzw. Lesezeiger in einer Datei an eine andere Position. Es gibt drei Modi: `"SEEK_SET"` geht an die angegebene absolute Position (Dateianfang ist Null), `"SEEK_CUR"` verschiebt den Zeiger um den Offset, und bei `"SEEK_END"` bezieht sich der Offset auf das Dateiende, also z. B. `"-1"` für das letzte Byte der Datei. Die Funktion liefert im Falle eines Fehlers einen von Null verschiedenen Wert.

```
SEEK_CUR, SEEK_SET, SEEK_END

#define SEEK_CUR 0
#define SEEK_END 1
#define SEEK_SET (-1)
```

Dies sind die drei Modi, die bei `"fseek"` angegeben werden können.

## ftell

```
long ftell(FILE *f)
```

liefert die aktuelle Dateiposition von Datei `"f"`.

## rewind

```
void rewind(FILE *f)
```

setzt die Datei `"f"` an den Anfang zurück und löscht das Fehlerflag. Damit ist `"rewind(f)"` eine Abkürzung für `"fseek(f, 0, SEEK_SET)"` mit anschließendem `"clearerr(f)"`.

## fpos_t

```
typedef int fpos_t
```

Der Datentyp `"fpos_t"` wird gebraucht, weil einige Implementierungen zwischen Text- und Binärdateien unterscheiden und die Anwendung `"fseek"` dann bei Textdateien etwas problematisch ist. Deshalb gibt es speziell für Textdateien die beiden folgenden Funktionen:

### fgetpos

```
int fgetpos(FILE *f, fpos_t *posn)
```

"fgetpos" speichert die aktuelle Position der Datei "f" in "posn", insbesondere zur späteren Verwendung mit "fsetpos". Wenn das System selbst nicht so genau weiß, was diese Posotion ist, liefert "fgetpos" einen Wert ungleich Null.

### fsetpos

```
int fsetpos(FILE *f, const fpos_t *posn)
```

Das versprochene Gegenstück zu "fgetpos": Die Datei wird wieder auf die Position gesetzt, die in "posn" mittels "fgetpos" abgespeichert wurde.

## 1.2.7 Sonstige Funktionen

### perror

```
void perror(const char *s)
```

gibt eine Fehlermeldung aus, die sich zum einen aus dem String "s" und zum anderen aus der Fehlermeldung, die zu der Nummer gehört, die gerade in "errno" steht. Hat "errno" z. B. den Wert 205, gibt

```
perror("Das war wohl nichts")
```

aus:

Das war wohl nichts: Object not found

"errno" wird eigentlich immer gesetzt, wenn irgend etwas schief geht, z. B. wenn eine Datei nicht geöffnet werden kann. Näheres steht in *<errno.h>*.

### exit

```
void exit(int res)
```

lesen Sie hierzu bitte in Kapitel 1.1.5 (stdlib.h) nach.

## 1.3 Fehlersuche: <assert.h>

### assert

```
#ifdef NDEBUG
#define assert(C)
#else
extern "Asm" void do__assert(char*, char*, unsigned int);
#define assert(C) { if(!(C)) do__assert(#C, __FILE__, __LINE__); }
#endif
```

Die Include-Datei **"assert.h"** definiert lediglich ein einziges Makro nämlich **"assert"**. Man kann damit die Sicherheit seines Programms erhöhen, indem man Plausibilitätstests einführt. Wenn z. B. eine Funktion einen Parameter zwischen 0 und 42 erwartet, setze man an ihren Anfang folgendes:

```
int f(int i)
{
assert( i>0 && i<42 )

i++;          // usw.
```

Gilt die bei **"assert"** angegebene Bedingung nicht, steigt das Programm bei einem ungültigen Parameter mit einer Meldung wie

```
Assertion failed: i>0 && i<42, file "murx.cpp", line 4711
```

aus - es sei denn, man definiert das Makro **"NDEBUG"**, bevor man es einbindet. In diesem Fall werden die **"assert"**-Anweisungen ersatzlos gestrichen und kosten so weder Laufzeit noch Speicherplatz. Man sollte **"NDEBUG"** immer erst dann definieren, wenn man der Meinung ist, das Programm sei fehlerfrei. Auf jeden Fall aber vermeide man Bedingungen mit Seiteneffekt:

```
FILE *fp;
assert(fp = fopen("Caddy.config", "r"))        // NEIN!
```

Diese Anweisung gibt zwar eine mehr oder weniger aussagekräftige Fehlermeldung ab, wenn das Programm seine Konfigurationsdatei nicht finden kann, und steigt dann aus, was durchaus sinnvoll ist - aber irgendwann kommt man vielleicht doch noch auf die Idee, **"NDEBUG"** zu definieren, und dann wird man sich wundern.

## 1.4 Zeichenklassen: <ctype.h>

In der Definitionsdatei *<ctype.h>* findet der Programmierer einige kleine, aber recht nützliche Funktionen, durch die Zeichen weitgehend unabhängig vom verwendeten Rechner und Zeichencode behandelt werden können. Dies ist enorm wichtig für saubere (d. h. portable) Programmentwicklung.

### isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

```
int isalnum (int c)
int isalpha (int c)
int iscntrl (int c)
int isdigit (int c)
int isgraph (int c)
int islower (int c)
int isprint (int c)
int ispunct (int c)
int isspace (int c)
int isupper (int c)
int isxdigit(int c)
```

Diese Funktionen testen, ob das Zeichen `"c"` zu einer bestimmten Zeichgenklasse gehört, und liefern dann den entsprechenden logischen Wert (also `"0"` oder `"1"`). Es gibt prinzipiell folgende Klassen von Zeichen:

- Großbuchstaben von `'A'` bis `'Z'`
- Kleinbuchstaben von `'a'` bis `'z'`
- Allgemeine Buchstaben, d. h. Groß- oder Kleinbuchstaben. Umlaute werden dabei jeweils leider nicht berücksichtigt.
- Dezimale Ziffern von `'0'` bis `'9'`
- Hexadezimale Ziffern: `'0'` bis `'9'`, `'A'` bis `'F'` sowie `'a'` bis `'f'`
- Leerraum, d. h. Leerzeichen, Zeilentrenner (Linefeed), Tabulatoren, Rücklauf (Return) und Seitenvorschub
- Sichtbare Zeichen einschließlich des Leerzeichens - das sind auf dem Amiga die Codes von `0x20` bis `0x7F` sowie von `0xA0` bis `0xFF`.
- Steuerzeichen, d. h. das genaue Gegenteil der sichtbaren Zeichen

Die folgende Liste gibt an, welche Funktion bei welcher Zeichenklasse „wahr" liefert:

isalnum Buchstabe oder Ziffer

| | |
|---|---|
| `isalpha` | Buchstabe |
| `iscntrl` | Steuerzeichen |
| `isdigit` | Dezimale Ziffer |
| `isgraph` | Sichtbares Zeichen außer Leerzeichen |
| `islower` | Kleinbuchstabe |
| `isprint` | Sichtbares Zeichen |
| `ispunct` | Sichtbares Zeichen außer Leerzeichen, Buchstabe oder Ziffer |
| `isspace` | Leerraum-Zeichen |
| `isupper` | Großbuchstabe |
| `isxdigit` | Hex-Ziffer |

## tolower

```
int tolower(int c)
```

Die Funktion wandelt das Zeichen `"c"`, sofern es sich dabei um einen Großbuchstaben handelt, in den entsprechenden Kleinbuchstaben um. Andernfalls wird das Argument unverändert zurückgegeben. Die Funktion entspricht dabei in etwa der Fallunterscheidung

```
if (c >= 'A' && c <= 'Z')
  return c+('a'-'A');
else
  return c;
```

Auch hier werden Umlaute leider nicht berücksichtigt.

## toupper

```
int toupper(int c)
```

Analog zu `"tolower"` werden hier Kleinbuchstaben in Großbuchstaben umgewandelt und andere Zeichen unverändert belassen.

## which_xdigit [M]

```
int which_xdigit(char c)
```

Wenn der Parameter `"c"` eine hexadezimale Ziffer ist, wird der zugehörige numerische Wert zurückgegeben, andernfalls `"-1"`. Beispiele:

```
which_xdigit('7')  = 7
```

```
which_xdigit('b')  = 11
which_xdigit('C')  = 12
which_xdigit('+')  = -1
```

## 1.5 Fehlernummern: <errno.h>

Während der Laufzeit eines Programms kann so einiges schiefgehen - ein guter Programmierer geht immer davon aus, daß alles schiefgeht, was irgendwie schiefgehen kann. Die Analyse von Laufzeitfehlern wird mit den Definitionen der Datei *<errno.h>* ein wenig erleichtert.

### errno

```
extern int errno
```

In dieser globalen Variablen legen viele Standard-Funktionen in Falle eines Falles eine Fehlernummer ab. Beispielsweise wird hier die entsprechende DOS-Fehlernummer zugewiesen, wenn bei `"fopen"` keine Datei geöffnet werden konnte. Vor solchen kritischen Operationen sollte man deshalb `"errno"` auf Null setzen.

### ERANGE

```
#define ERANGE 1000
```

Diese Fehlernummer wird `"errno"` zugewiesen, wenn bei einer Funktion ein Wertebereich überschritten wurde.

### EUSRBRK [M]

```
#define EUSRBRK 900
```

Diese Konstante gibt den Rückgabewert eines Programms an, wenn es vom Benutzer abgebrochen wurde. Obwohl die Konstante in *<errno.h>* definiert wird, wird dieser Wert dabei nicht wirklich an `"errno"` zugewiesen und kann folglich auch nicht abgefragt werden. Vielmehr wird diese Konstante definiert, damit man mit einer Anweisung wie `"exit(EUSRBRK)"` ein derartiges Programmende simulieren kann.

## EASSERT [M]

```
#define EASSERT 990
```

Genau wie "EUSRBRK" ist dies kein Wert, den "errno" annimmt, sondern der Rückgabewert eines Programms, und zwar dann, wenn bei einem "assert"-Makro ein Fehler entdeckt wird.

## EFREEMEM [M]

```
#define EFREEMEM 996
```

Auch dies ist ein Rückgabewert und keine wirkliche Fehlernummer. Mit diesem Wert bricht ein Programm ab, wenn bei "free" bzw. "delete" ein ungültiger Speicherbereich freigegeben werden soll.

## ENONUM [M]

```
#define ENONUM 1001
```

Dieser Fehlercode wird von Funktionen wie "atoi" oder "strtod" gesetzt, wenn ein String keine gültige Zahldarstellung ist.

## ENOMEM [M]

```
#define ENOMEM 1002
```

"new" und die "malloc"-Funktion setzen diese Fehlernummer, wenn kein Speicher reserviert werden konnte.

## 1.6 Ganzzahlige Grenzwerte: <limits.h>

Jede Implementierung kann mehr oder weniger frei festlegen, welchen Umfang die einzelnen Datentypen haben. In der Datei *<limits.h>* werden Konstanten definiert, die die spezifischen Daten eines C-Systems angeben und so standard-gemäße Programmierung erleichtern.

### CHAR_BIT

```
#define CHAR_BIT 8
```

gibt die Anzahl der Bits in einem `"char"` an.

### CHAR_MAX, CHAR_MIN, SCHAR_MAX, SCHAR_MIN, SHRT_MAX SHRT_MIN, INT_MAX, INT_MIN, LONG_MAX, LONG_MIN, LONGLONG_MAX, LONGLONG_MIN

```
#define CHAR_MAX        127
#define CHAR_MIN        (-128)
#define SCHAR_MAX       127
#define SCHAR_MIN       (-128)
#define SHRT_MAX        0x7fff
#define SHRT_MIN        (-0x8000)
#define INT_MAX         0x7fffffff
#define INT_MIN         (-0x80000000)
#define LONG_MAX        0x7fffffff
#define LONG_MIN        (-0x80000000)
#define LONGLONG_MAX    0x7fffffffffffffffLL
#define LONGLONG_MIN    (-0x8000000000000000LL)
```

Zu jedem vorzeichenbehafteten Datentyp existieren je zwei Konstanten, die den minimalen bzw. maximalen Wert dieses Typs angeben. Dementsprechend enden die Namen dieser Konstanten auf `"_MIN"` oder `"_MAX"` und beginnen je nach Typ mit `"CHAR"`, `"SCHAR"`, `"SHRT"`, `"INT"`, `"LONG"` oder `"LONGLONG"`.

## UCHAR_MAX, USHRT_MAX, UINT_MAX, ULONG_MAX, ULONGLONG_MAX

```
#defineUCHAR_MAX 255
#defineUSHRT_MAX 65535
#defineUINT_MAX  0xffffffffU
#defineULONG_MAX 0xffffffffU
#defineULONGLONG_MAX  0xffffffffffffffffULL
```

Bei den vorzeichenbehafteten ganzzahligen Datentypen ist naturgemäß stets Null der kleinste darstellbare Wert. Deshalb existiert für diese Typen lediglich jeweils eine Konstante, die den höchsten Wert des Datentyps darstellt.

# 1.7 Mathematische Funktionen: <math.h>

## *1.7.1 Umwandlung von Zahlen in Zeichenketten*

## PARAMETER_BASE, PARAMETER_DIGITS [M]

```
#ifdef __cplusplus
#define PARAMETER_BASE short=10
#define PARAMETER_DIGITS short=0
#else
#define PARAMETER_BASE short
#define PARAMETER_DIGITS short
#endif
```

Die nachfolgend beschriebenen Funktionen sind allesamt Spezialitäten von MaxonC++ und wurden durch Default-Argumente etwas komfortabler gemacht. Da ANSI C aber keine Default-Argumente kennt, werden in den Parameterlisten diese beiden Makros benutzt. In C++ kann man also das jeweilige Argument weglassen, in C muß man es setzen. Übrigens werden die Makronamen am Ende von *<math.h>* mit `"#undef"` wieder gelöscht, so daß im weiteren keine Kollisionen auftreten.

## inttostr, uinttostr, vlongtostr, uvlongtostr [M]

```
char *inttostr (int i, char s[ ], PARAMETER_BASE)
char *uinttostr (unsigned u, char s[ ], PARAMETER_BASE)
char *vlongtostr (long long ll, char s[ ], PARAMETER_BASE)
char *uvlongtostr (unsigned long long ull, char s[ ], PARAMETER_BASE)
```

Diese vier Funktionen wandeln ihr jeweils erstes Argument in eineZeichenkette um und legen das Ergebnis im String `"s"` ab. Der (in C++ optionale) dritte Parameter gibt die gewünschte Basis, eine Zahl von 2 bis 36, an, wobei 10 Default-Wert ist. Die umgewandelte Zeichenkette wird natürlich mit einem Nullzeichen abgeschlossen und ein Zeiger auf `"s"` zurückgegeben. Sie als Programmierer sind selbst dafür verantwortlich, daß der String `"s"` lang genug für das Ergebnis ist.

## doubletostr, floattostr [M]

```
char *floattostr (float f, char s[ ], PARAMETER_DIGITS)
char *doubletostr (double d, char s[ ], PARAMETER_DIGITS)
```

Mit diesen beiden Funktionen können Sie bequem und ohne Benutzung von `"sprintf"` eine Fließkommazahl in eine Zeichenkette verwandeln. Das Ergebnis mit einem abschließenden Nullbyte wird dabei im String `"s"` abgelegt (der natürlich hinreichend lang sein sollte), und der letzte Parameter, für den im C++-Modus der Default-Wert Null definiert ist, gibt den Modus und die Ziffernanzahl an:

Bei einer positiven Anzahl von Stellen entspricht diese der Zahl der Nachkommastellen:

```
doubletostr(17.4, s, 5)
```

liefert

**s = " 17.40000",**

während eine negative Anzahl die Gesamtzahl der Dezimalstellen bei Exponentialdarstellung angibt:

```
doubletostr(17.4, s, -5)
```

führt zum Ergebnis

**s = " 1.7400e+001".**

und beim Argument `"0"` (bzw. dem Default-Argument) wird für Zahlen zwischen 10000 und 0.1 die Festpunktdarstellung mit der minimalen Anzahl von Nachkommastellen und sonst die Exponentialdarstellung gewählt.

Im Gegensatz zu den zuvor beschriebenen ganzzahligen Formatierungsfunktionen wird bei `"doubletostr"` und `"floattostr"` positiven Zahlen ein Leerzeichen vorangestellt.

## 1.7.2 Fließkomma-Berechnungen

### sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exp, log, log10, sqrt

```
double sin (double x)
double cos (double x)
double tan (double x)
double asin (double x)
double acos (double x)
double atan (double x)
double sinh (double x)
double cosh (double x)
double tanh (double x)
double exp (double x)
double log (double x)
double log10 (double x)
double sqrt (double x)
```

Diese Funktionen stellen die wichtigsten mathematischen Funktionen dar. Offensichtlich sind ihr Argument und Ergebnis jeweils `"double"`-Werte. Die Funktionen haben folgende (meist offensichtliche) Bedeutung:

| | |
|---|---|
| `sin` | Sinus |
| `cos` | Cosinus |
| `tan` | Tangens |
| `asin` | sin ^ -1 für Argumente zwischen -1 und 1 |
| `acos` | cos ^ -1 für Argumente zwischen -1 und 1 |
| `atan` | tan ^ -1, Ergebnis zwischen -pi/2 und +pi/2 |
| `sinh` | Sinus Hyperbolicus |
| `cosh` | Cosinus Hyperbolicus |
| `tanh` | Tangens Hyperbolicus |
| `exp` | Exponentialfunktion e ^ x |
| `log` | Natürlicher Logarithmus |
| `log10` | Logarithmus zur Basis 10 |
| `sqrt` | Quadratwurzel |

Bei den trigonometrichen Funktionen werden Winkel natürlich im Bogenmaß (Radian) angegeben. Ein Gradwinkel `"w"` läßt sich mit der Formel

```
x = 3.14159265358979/180.0 * w
```

in Bogenmaß umrechnen.

## pow

```
double pow (double x, double y)
```

Diese Funktion berechnet die Potenz x ^ y, wobei aber `"x"` positiv sein muß.

## atan2

```
atan2(double x, double y)
```

berechnet den Arcustangens von `y/x`, d. h. den Winkel, der zum Cosinus `"x"` und dem Sinus `"y"` gehört. Das Ergebnis ist aus `[-pi, +pi]`, und die Funktion ist besonders dann praktisch, wenn man kartesische in Polarkoordinaten umrechnen will.

## floor

```
floor(double x)
```

Die Funktion `"floor"` rundet auf die nächstkleinere ganze Zahl ab (genaugenommen die größte ganze Zahl kleiner oder gleich x). Das Ergebnis ist nichtsdestotrotz ein `"double"`- und kein `"int"`-Wert.

## ceil

```
double ceil(double x)
```

Analog zu `"floor"` rundet `"ceil"` ihr Argument auf.

## fabs

```
fabs(double x)
```

`"fabs"` liefert den Betrag ihres Arguments `"x"`.

## ldexp

```
ldexp(double x,int n)
```

Mit dieser Funktion erzeugt man aus einer Mantisse `"x"` und einem Binär-Exponenten `"n"` eine Fließkommazahl `"x * 2^n"`. `"ldexp"` ist vor allem dann nützlich und auch ausgesprochen schnell, wenn man einen `"double"`-Wert mit einer Zweierpotenz multiplizieren will.

## frexp

```
double frexp(double x ,int *expo)
```

`"frexp"` ist das Gegenstück zu `"ldexp"`, denn hier wird das Argument in eine normalisierte Mantisse im Bereich von 0.5 bis 1 und einen Exponenten zur Basis 2 zerlegt. Die Mantisse wird als Funktionswert zurückgegeben, während der Exponent in `"*expo"` abgelegt wird.

## modf

```
double modf(double x, double *dp)
```

Die Zahl `"x"` wird in ihre Vor- und Nachkommastellen aufgeteilt. Die Nachkommastellen, also eine Zahl zwischen -1 und +1, sind das Resultat der Funktion, während der ganzzahlige Teil nach `"*dp"` geschrieben wird. Beide Ergebnisse haben das gleiche Vorzeichen wie `"x"`.

## fmod

```
double fmod(double x, double y)
```

Diese Funktion sollte nicht mit `"modf"` verwechselt werden, sie berechnet den Rest, der bei einer Fließkommadivision `"x/y"` bei ganzzahligem Quotienten auftritt.

## pwr10 [M]

```
double pwr10(int i)
```

Für ganzzahlige `"i"` im Bereich von `-308` bis `+308` liefert `"prw10"` die Potenz `10^i`, und zwar ziemlich schnell, denn die Werte werden dabei teilweise einer Tabelle entnommen.

## expo10 [M]

```
int expo10(double x)
```

Diese Funktion liefert die größte ganze Zahl "n", für die "prw10(n)" nicht größer als "x" ist, also den Exponententeil von "x". Nicht ganz zufällig sind "pwr10" und "expo10" Abfallprodukte der Stringkonvertierungsfunktionen von Fließkommazahlen.

## fpwr10 [M]

```
float fpwr10(int i)
```

"fpwr10" ist das "float"-Gegenstück zu "pwr10" und berechnet ebenfalls die Potenz "10^i", aber als "float"-Wert.

# 1.8 Haarsträubende Sprünge: <setjmp.h>

## setjmp, longjmp

```
int setjmp(jmp_buf buf)
void longjmp(jmp_buf buf, int num)
```

Die normalen "goto"-Sprünge können nur innerhalb einer Funktion herumspringen, und das ist oft nicht genug. Manchmal muß man von einer Funktion in eine andere zurückspringen, z. B. wenn irgendwo in einer tief verschachtelten Reihe von Funktionsaufrufen ein Fehler auftritt und die Programmfunktion, aber nicht das ganze Programm abgebrochen werden soll. Es wäre dabei meist sehr umständlich, hinter jeden der zahllosen Funktionsaufrufe eine Abfrage der Art „Ist-ein-Fehler- aufgetreten-wenn-ja-dann-Ende" zu setzen. Deshalb gibt es die beiden Funktionen "setjmp" und "longjmp", mit denen man direkt aus einer Funktion in eine andere, höher gelegene zurückspringen kann.

Man kann dabei keineswegs wild in irgendeine Funktion hineinspringen, denn dann wäre der Absturz nahezu garantiert. Vielmehr speichert "setjmp" die wichtigsten Daten eines Zustands, nämlich den Stackpointer, den Programmzähler und die übrigen Prozessorregister, in einem Puffer des Typs "jmp_buf" ab, und "longjmp" stellt eben diesen Zustand wieder her, aber nur dann, wenn die Funktion, in der "setjmp" aufgerufen wurde, bisher nicht beendet wurde.

Der Ablauf ist also immer der folgende: In einer Funktion "f1" wird "setjmp" aufgerufen, und damit merkt sich das Programm haargenau die Position dieses "setjmp"-Aufrufs. Nun kann aus "f1" eine andere Funktion "f2" aufgerufen werden, welche möglicherweise wiederum irgendwann eine Funktion "f3" aufruft, in der ein Aufruf von "f4" stattfindet... Jedenfalls

befindet das Programm sich irgendwann in einer Funktion `"fx"`, und dann wird mit `"longjmp"` der abgespeicherte Zustand wiederhergestellt: Die Funktion `"fx"` wird beendet und alle ihre Daten sowie die der anderen Funktionen bis einschließlich `"f2"` werden vom Stack entfernt. Die Programmausführung wird in `"f1"` an genau der Stelle, an der das `"setjmp"` steht, fortgesetzt.

Nach jedem `"setjmp"` sollte das Programm dann feststellen, was passiert ist: Wurde soeben `"setjmp"` ausgeführt und der Programmzustand abgespeichert, oder ist ein `"longjmp"` erfolgt? Im letzteren Fall befindet sich das Programm schließlich scheinbar auch direkt hinter dem Aufruf von `"setjmp"`. Deshalb hat `"setjmp"` einen Ergebniswert, der im Prinzip immer Null ist, und der Funktion `"longjmp"` kann ein numerisches Argument übergeben werden. Beim `"longjmp"`-Sprung wird dem Programm dann vorgegaukelt, `"setjmp"` habe dieses Argument als Ergebnis geliefert.

Am besten wird die Sache wohl an einem Beispiel deutlich. Eine typische Anwendung von `"setjmp"` und `"longjmp"` sieht so aus:

```
#include <setjmp.h>

jmp_buf Ausstieg;

void f2()
{
  printf("Funktion f2:\n");

  longjmp(Ausstieg,17);

  printf("PANIC\n");     // Diese Stelle wird nie erreicht.
}

void f1()
{
  if(setjmp(Ausstieg) == 0)
  // Ergebnis 0: Es wurde wirklich bloß der Zustand gespeichert
  {
    printf("Jump gesetzt.\n");
    f2();
    // Diese Stelle wird nur erreicht,
    // wenn kein "longjmp" gemacht wird:
    printf("DOUBLE PANIC\n");
  }
  else
    // Ergebnis 17: longjmp wurde ausgeführt
    printf("Jump ausgeführt.\n");
}
```

Die Funktion `"f1"` macht dabei folgende Ausgabe:

**Jump gesetzt.**

**Funktion f2:**

**Jump ausgeführt.**

`"jmp_buf"` ist stets ein Vektortyp, weshalb bei `"setjmp"` kein Adressoperator `"&"` vor das Argument gesetzt werden muß. Ihnen dürfte (oder sollte) klar sein, daß diese Funktion in der Regel ziemlich hanebüchen implementiert wird. Deshalb darf man `"setjmp"` nur in ganz bestimmten Zusammenhängen benutzen:

- Man darf auf `"setjmp"` unäre Operatoren, z. B. `"-"` oder `"!"`, anwenden.
- Das Ergebnis kann mit den Operatoren `"=="`, `"!="`, `"<"`, `">"`, `"<="` oder `">="` mit einem konstanten numerischen Ausdruck verglichen werden.
- Der Ausdruck kann „einfach so" als Anweisung stehen oder bei `"if"`, `"while"`, `"do"` oder `"switch"` als Bedingung verwendet werden.

Ein Ausdruck wie

```
printf("Ergebnis: %d\n", setjmp(x))
```

würde also unter den meisten Implementierungen (einschließlich MaxonC++) beim anschließenden `"longjmp"` zu einem gnadenlosen Absturz führen. Selbst ein scheinbar so harmloses Konstrukt wie

```
int i;
i = setjmp(x);
if (i == 0)
{ usw.
```

ist nicht zulässig (auch wenn es bei MaxonC++ zufällig klappt).

Es gibt noch einige andere Einschränkungen: Da `"longjmp"` ein echtes Low-Level-Konstrukt ist, das den Compiler praktisch heimtückisch überlistet, werden zwar die automatischen Variablen der solchermaßen „abgewürgten" Funktionen vom Stack entfernt, aber keine Destruktoren aufgerufen. Nach der Rückkehr durch `"longjmp"` müssen Sie auch damit rechnen, daß die Optimierungen des Compilers Ärger machen und automatische Variablen jener Funktion, die nicht als `"volatile"` deklariert sind, undefinierte Werte haben.

## jmp_buf

```
typedef int jmp_buf[_JMP_BUF_SIZE]
```

Dieser Datentyp repräsentiert einen geeigneten Puffer, in dem `"setjmp"` einen Programmzustand abspeichern kann.

## JMP_BUF_SIZE [M]

```
#define _JMP_BUF_SIZE 16
```

Eine kleine, unbedeutende Hilfskonstante, die die Größe des Vektortyps `"jmp_buf"` angibt.

# 1.9 Signale und Ereignisse: <signal.h>

## signal

```
void (*signal(int sig, void(*f)(int)))(int)
```

oder besser:

```
typedef void (*P2F)(int)
P2F signal (int sig, P2F f)
```

Die Funktion `"signal"` setzt eine Funktion, die aufgerufen werden soll, wenn ein bestimmtes Signal `"sig"` eintrifft. In der vorliegenden Version von MaxonC++ ist die Funktion `"signal"` weitgehend Makulatur, denn bisher wird ein Signal ausschließlich vom Programm selbst durch die Funktion `"raise"` gesendet. Der Parameter `"sig"` gibt die jeweilige Signalnummer - in der Regel eines der im folgenden beschriebenen Makros - an, und `"f"` ist ein Zeiger auf eine Funktion, die ein `"int"` als Argument erhält und kein Ergebnis (also `"void"`) liefert. Das Ergebnis von `"signal"` ist ein Zeiger auf die Funktion, die vorher als Signal-Handler gesetzt war.

Anschließend wird die Funktion `"f"` stets beim Eintreffen eines Signals `"sig"` aufgerufen und erhält dabei als Argument die Signalnummer, so daß eine Funktion durchaus als Handler für mehrere Signale verwendet werden kann.

Anstelle eines Zeigers auf eine Funktion kann auch eins der unten beschriebenen Makros `"SIG_IGN"`, `"SIG_DFL"` oder `"SIG_ERR"` verwendet werden.

## raise

```
int raise(int sig)
```

Die Funktion `"raise"` sendet das Signal `"sig"` und löst so den Aufruf des entsprechenden Handlers aus.

## SIGTERM, SIGABRT, SIGPFE, SIGILL, SIGINT, SIGSEGV

```
#defineSIGTERM0  // Normales Programmende
#defineSIGABRT1  // Anormales Programmende wie bei "abort"
#defineSIGFPE 2  // Arithmetikfehler
#defineSIGILL 3  // Illegale Prozessor-Anweisung
#defineSIGINT 4  // Software-Interrupt
#defineSIGSEGV5  // Segmentverletzung bei Speicherzugriff
```

Dies sind die sechs Standard-Signalnummern und damit die einzigen wirklich legalen Argumente für `"signal"` und `"raise"`. Eine Implementierung darf aber auch noch zusätzliche eigene Signale haben.

## SIG_IGN

```
#define SIG_IGN ((void(*)(int))0)
```

Wird bei `"signal"` als Handlerfunktion `"SIG_IGN"` angegeben, so wird das entsprechede Signal in Zukunft ignoriert.

## SIG_DFL

```
#define SIG_DFL ((void(*)(int))1)
```

Diese Pseudo-Handlerfunktion stellt als Argument bei `"signal"` den jeweiligen Default-Handler dar.

## SIG_ERR

```
#define SIG_ERR ((void(*)(int))-1)
```

Die Funktion `"signal"` liefert das Ergebnis `"SIG_ERR"`, wenn irgendein Fehler aufgetreten ist, z. B. die Argumente illegal waren.

## sig_atomic_t

```
typedef int sig_atomic_t
```

Für echte Signal-Freaks gibt es den ganzzahligen Datentyp **"sig_atomic_t"**, der zur Kommunikation zwischen Programm und Handler benutzt werden kann. Die „besondere" Eigenschaft dieses Datentyps ist, daß ein Zugriff auf eine solche Variable eine unteilbare Operation ist, d. h. nicht unterbrochen werden kann. Wenn man einer Variablen dieses Typs einen Wert zuweist, kann man darauf vertrauen, daß ein währenddessen eintreffendes Signal entweder vor oder nach dieser Operation ausgeführt wird und nicht etwa mittendrin, wenn die Variable so etwas wie einen Zwischenwert hat.

Oder drücken wir es einmal so aus: Eine **"long long"**-Variable besteht unter MaxonC++ aus zwei Langworten, die bei einer Wertzuweisung nacheinander geschrieben werden. Hier könnte es durchaus vorkommen, daß ein Signal-Handler genau dann aufgerufen wird, wenn erst eines der beiden Langworte geschrieben ist. Der Wert der Variablen wäre dann während der Ausführung des Handlers irgendwie eine undefinierte Mischung aus dem alten und dem neuen Wert. Wenn man also in einer globalen Variablen Parameter an einen Handler übergeben möchte, sollte man dafür ausschließlich den Datentyp **"sig_atomic_t"** benutzen.

# 1.10 Variable Argumentlisten: <stdarg.h>

## va_list

```
typedef unsigned int va_list
```

Funktionen wie **"printf"** und **"scanf"** haben eine Parameterliste, in der jeweils nur die ersten Parameter einen Namen und Typ haben und anschließend beliebig viele weitere Argumente erlaubt sind. Syntaktisch wird dies in C++ durch die Ellipse **"..."** am Ende der Parameterliste deklariert, und man kann dieses Feature auch in selbstgeschriebenen Funktionen verwenden. Nun wird sich der geneigte Leser vielleicht fragen, wie eine Funktion denn solche namenlose Parameter mit nahezu beliebiger Anzahl und unterschiedlichsten Typen auswerten kann. Genau dabei hilft die Includedatei „*<stdarg.h>*", die einige Makros deklariert, mit denen man Stack-Argumente gleichermaßen komfortabel und portabel auswerten kann.

Das Prinzip beruht darauf, daß ein Zeiger hinter den letzten normal deklarierten Parameter gesetzt wird und dann der Reihe nach die restliche Argumentliste durchläuft. Dieser „Zeiger" wird repräsentiert durch eine Variable des Typs **"va_list"**.

## va_start

```
#define va_start(AP,LASTARG) (AP)=(unsigned int)(&LASTARG) \
+sizeof(LASTARG)
```

Ein Argumentlisten-Zeiger des Typs `"va_list"` wird normalerweise initialisiert, indem er auf die erste Adresse hinter dem letzten Parameter gesetzt wird, also auf genau die Speicheradresse, an der die formal unbekannten, restlichen Argumente anfangen. Dazu dient das Makro `"va_start"`, z. B. so:

```
void fun(const char *format...)
{
  va_list v;
  va_start(v, format);

  // usw.
}
```

## va_arg

```
#define va_arg(AP,TYPE) ((AP)+=sizeof(TYPE), \
sizeof(TYPE)>1 ? ((AP)=(AP)+1&(-2)):(AP), \
((TYPE*)AP)[-1])
```

Das Makro `"va_arg"` mit seiner ziemlich wüsten Definition ist in seiner Anwendung um so einfacher: Ein Aufruf der Form

```
x = va_arg(vp, TYPE);
```

liest das nächste Argument des Typs `"TYPE"` aus der variablen Argumentliste, liefert es als Ergebnis und setzt nebenbei den Argumentzeiger `"vp"` entsprechend weiter. Wenn man also die auftretenden Datentypen kennt (bei `"printf"` gibt man diese ja im Formatstring an), kann man mit einer Folge von `"va_arg"`-Aufrufen nacheinander die Argumente lesen.

Beachten Sie aber, daß „kurze" ganzzahlige Typen (short, char) bei dieser Art der Wertübergabe automatisch nach `"int"` sowie `"float"` nach `"double"` umgewandelt werden. Deshalb ist z. B. `"va_arg(x, unsigned short)"` generell unsinnig und ist durch `"va_arg(x, unsigned int)"` zu ersetzen.

## va_end

```
#define va_end(AP)
```

Bei einigen Implementierungen kann die Auswertung variabler Argumentlisten ein ziemicher Akt sein, und deshalb gibt es das Makro `"va_end"`, mit dem man immer eine Argumentauswertung beenden sollte und das bei Bedarf irgendwelche Aufräumarbeiten durchführt. Bei MaxonC++ ist `"va_end"` ein leeres Makro, aber trotzdem sollte man es nie vergessen.

Zum Abschluß ein zusammenhängendes Beispiel für die Auswertung variabler Argumentlisten. Der Einfachheit halber ist die Argumentliste hier überhaupt nicht variabel, sondern es wird stets erwartet, daß ein `"int"`-, ein `"double"`- und ein `"char"`-Argument folgen. Trotzdem sollte das Prinzip klar werden:

```
#include <stdarg.h>
#include <stdio.h>

void fun(const char *format...)
{
  va_list v;
  int i;
  double d;
  char c;

  va_start(v, format);  // initialisieren mit letztem bekannten
  // Parameter

  // die drei Argumente werden der Reihe nach gele     sen:
  i = va_arg(v, int);
  d = va_arg(v, double);
  c = (char) va_arg(v, int); // Auch chars werden als "int"
        // Übergeben

  // der Ordnung halber aufräumen:
  va_end(v);

  printf("%s: %i , %g , %c\n", format, i, d, c);
}

void main()
{
  fun("Test", 42, 47.11, 'x');
}
```

# 1.11 Implementationsabhängige Definitionen: <stddef.h>

## NULL

```
#define NULL 0
```

Die allgemein beliebte und geschätzte Konstante `"NULL"` wird auch in „*<stddef.h>*“ deklariert.

## offsetof

```
#define offsetof(s,m) ((unsigned)&((s*)NULL)->m)
```

In ANSI C gibt es keine Pointer auf Member. Als die C-Programmierer daraufhin neidisch nach C++ schielten, kamen sie offensichtlich auf die Idee, dieses Feature in C zu simulieren, indem sie den Offset eines Members innerhalb einer Struktur berechnen. Mit dem Makro `"offsetof"` wird dies erheblich erleichtert: `"offsetof(s,m)"` liefert den Offset (in Bytes) des Members `"m"` in der Klasse oder Struktur `"s"`.

## size_t

```
typedef unsigned size_t
```

`"size_t"` wird nicht nur in „*<stdlib.h>*“ und „*<stdio.h>*“, sondern auch in `"stddef.h"` definiert und stellt den Typen dar, den das Ergebnis von `"sizeof"` in einer bestimmten Implementation hat.

## ptrdiff_t

```
typedef int ptrdiff_t
```

Subtrahiert man zwei Zeiger gleichen Typs voneinander, so ist das Ergebnis bekanntlich ein ganzzahliger Wert, der die Anzahl der Vektorelemente zwischen diesen Zeigern angibt. Nun kann der Typ dieser Differenz durchaus implementationsabhängig sein: Bei einigen Compilern muß eine Adressdistanz als `"long int"` ausgedrückt werden, bei anderen (wie MaxonC++) reicht auch ein `"int"`. Deshalb gibt es für diesen Typ den standardisierten Namen `"ptrdiff_t"`.

### wchar_t

```
typedef int wchar_t
```

Noch eine Wiederholung: Der Name `"wchar_t"` bezeichnet den Datentyp von langen Zeichenkonstanten.

## 1.12 Zeichenketten und Speicherverwaltung: <string.h>

### strcpy

```
char *strcpy (char *s1, const char *s2)
```

Das Vektorkonzept von ANSI C ist bekanntlich „far beyond repair" und hat unter anderem zur Folge, daß man Strings nicht mit einem einfachen `"="` zuweisen kann. Deshalb gibt es die überaus nützliche Funktion `"strcpy"`, die einen String `"s2"`, der mit einem Nullbyte enden sollte, in eine Stringvariable `"s1"` kopiert. Das Ergebnis ist undefiniert, wenn die beiden Strings sich überlappen, und wie eigentlich immer haben Sie höchstpersönlich darüber zu wachen, daß die Zielvariable für den String `"s2"` lang genug ist. Das Ergebnis von `"strcpy"` ist der Zeiger `"s1"`.

### strncpy

```
char *strncpy(char *s1, const char *s1, size_t n)
```

Die sicherheitsbetonte Variante von `"strcpy"` kopiert maximal `"n"` Zeichen aus `"s2"` nach `"s1"` und füllt den Rest mit Nullzeichen auf.

### strcat

```
char *strcat (char *s1, const char *s2)
```

In diesem Fall heißt `"cat"` nicht „Katze", sondern ist eine Verbalhornung von `"concat"` oder auf Deutsch **„konkatinieren"** - man kann auch „aneinanderhängen" dazu sagen. Die Funktion sucht sich das Ende des Zeichenvektors `"s1"` (am obligatorischen Nullbyte unschwer zu erkennen) und kopiert den String `"s2"` genau dorthin, wodurch die beiden Strings zusammengehängt werden. Auch hier ist es der Sorgfalt und Umsicht des Programmierers anheim gestellt, dafür zu sorgen, daß im String `"s1"` auch noch genug Platz ist. Der Rückgabewert ist wieder einmal ein Zeiger auf das Ergebnis, also `"s1"`.

## strncat

```
char *strncat(char *s1, const char *s2, size_t n)
```

Genau wie bei `"strcat"` wird auch hier der String `"s2"` an den Inhalt des Zeichenvektors `"s1"` angehänht, aber hier werden maximal `"n"` Zeichen kopiert.

## strcmp

```
int strcmp (const char *s1, const char *s2)
```

Diese Funktion vergleicht die beiden Zeichenketten `"s1"` und `"s2"` und liefert:

- einen negativen Wert, wenn `s1 < s2`,
- Null, falls `s1 == s2`,
- eine positives Ergebnis, sofern `s1 > s2`.

Natürlich werden hier wirklich die Zeichen des Strings verglichen und nicht etwa die Adressen, wie es ja bei gewöhnlichen Stringvergleichen der Fall ist.

## strncmp

```
int strncmp(const char *s1, const char *s2, size_t n)
```

`"strncmp"` entspricht `"strcmp"`, vergleicht aber maximal die ersten `"n"` Zeichen, oder auch weniger, wenn eher auf ein Nullzeichen und damit das Stringende gestoßen wird.

## stricmp [M]

```
int stricmp(const char *s1, const char *s2)
```

Diese Funktion vergleicht die Zeichenketten `"s1"` und `"s2"` genau wie `"strcmp"`, wandelt aber zuerst alle Klein- in Großbuchstaben um, bevor zwei Zeichen verglichen werden. Diese Umwandlung geschieht natürlich nur intern, so daß es keinen Seiteneffekt auf die Strings `"s1"` und `"s2"` gibt. Beispielsweise liefert `"stricmp("Test", "TEST")"` Null.

## strlwr [M]

```
char *strlwr(char *s)
```

In der durch ein Nullzeichen abgeschlossenen Zeichenkette `"s"` werden alle Großbuchstaben durch Kleinbuchstaben ersetzt. Ein Zeiger auf den solchermaßen veränderten String wird zurückgegeben.

## strupr [M]

```
char *strupr(char *s)
```

funktioniert wie `"strlwr"`, wandelt aber Klein- in Großbuschtaben um.

## strchr

```
char *strchr (const char *s, int c)
```

Die Funktion sucht das erste Zeichen `"c"` im String `"s"` und liefert einen Zeiger darauf. Sollte in `"s"` kein `"c"` vorkommen, ist das Ergebnis Null.

## strrchr

```
char *strrchr(const char *s, int c)
```

leifert analog zu `"strchr"` einen Zeiger auf das letzte `"c"` in `"s"` oder Null, falls es so etwas nicht gibt.

## strspn

```
size_t strspn (const char *s, const char *c)
```

liefert die Anzahl der Zeichen am Anfang von `"s"`, die alle auch in `"c"` vorkommen, z. B.

```
strspn("caddy", "abcd") == 4
```

## strcspn

```
size_t strcspn(const char *s, const char *c)
```

ermittelt die Anzahl der Zeichen am Anfang von `"s"`, die allesamt nicht im String `"c"` vorkommen.

## strpbrk

```
char *strpbrk(const char *s, const char *c)
```

ist eine Art erweiterte Version von `"strchr"` und liefert einen Zeiger auf die erste Stelle in `"s"`, an der irgendein Zeichen aus `"c"` vorkommt.

## strstr

```
char *strstr(const char *s1, const char *s2)
```

Diese Funktion sucht die komplette Zeichenkette `"s2"` im String `"s1"` und liefert in gewohnter Manier einen Zeiger auf ihr erstes Auftreten oder Null.

## strlen

```
size_t strlen(const char *s)
```

liefert die Länge des Strings `"s"`, wobei das Nullzeichen am Ende nicht mitgerechnet wird.

## strerror

```
char *strerror(int n)
```

liefert die Fehlermeldung zur Fehlernummer `"n"`, also im Wesentlichen das, was `"perror"` ausgibt.

## strtok

```
char *strtok(char *s, const char *c)
```

Mit der Funktion "strtok" kann man eine Zeichenkette "s" in eine Folge von Zeichenketten, die jeweils durch Zeichen aus "c" begrenzt sind, zerlegen. Beim ersten Aufruf ist für "s" ein Zeiger auf den Anfang der Zeichenkette zu setzen, bei allen weiteren ist hier "Null" zu setzen. Die Funktion sucht jeweils die erste Position im String, an der ein Zeichen aus "c" auftritt, setzt an diese Stelle ein '\0' und gibt einen Zeiger auf den Anfang dieses Teilstrings zurück. Dabei merkt sie sich die Position, an der das Zeichen aus "c" gefunden und das Nullzeichen gesetzt wurde, und sucht beim nächsten Aufruf, sofern als erstes Argument Null übergeben wird, von dieser Stelle weiter.

Wenn auf diese Weise der gesamte String verhackstückt wurde und kein Zeichen aus "c" mehr gefunden wurde, liefert "strtok" das Ergebnis Null.

Der sittliche Nährwert dieser Funktion liegt darin, eine Zeichenfolge "s" in eine Folge von Token, die durch Trennzeichen aus "c" separiert werden, zu zerbröseln. Beispielsweise zerlegt das folgende Programm einen String in Worte, wobei eine Zeichenfolge, die keine Leerzeichen oder Zeilentrenner enthält, als Wort betrachtet wird:

```
#include <string.h>
#include <stdio.h>

void main()
{
  char s[ ] = "Alles Seiende entsteht ohne Grund,\nsetzt sich aus
  Schwäche fort und stirbt durch Zufall.", trenn[ ] = " \n";
  char *pos;
  int i = 0;

  // erstes Wort abtrennen:
  pos = strtok(s,trenn);

  while(pos) // d. h, solange noch Worte gefunden werden:
  {
    // Wort ausgeben:
    printf("Wort %2i: %s\n", ++i, pos);

    // nächstes Wort ermitteln:
    pos = strtok(0,trenn);
  }
}
```

Das Programm liefert folgende Ausgabe:

```
Wort 1: Alles
Wort 2: Seiende
Wort 3: entsteht
Wort 4: ohne
Wort 5: Grund,
Wort 6: setzt
Wort 7: sich
Wort 8: aus
Wort 9: Schwäche
Wort 10: fort
Wort 11: und
Wort 12: stirbt
Wort 13: durch
Wort 14: Zufall.
```

## memcpy

```
void *memcpy(void *v1, const void *v2, size_t n)
```

Von Adresse `"v2"` werden `"n"` Bytes an Adresse `"v1"` kopiert. Dabei dürfen sich die beiden Speicherbereiche nicht überlappen. Ergebnis der Funktion ist `"v1"`.

## memmove

```
void *memmove(void *v1, const void *v2, size_t n)
```

entspricht `"memcpy"`, aber hier dürfen sich der Quell- und der Zielbereich auch überlappen.

### memcmp

```
int memcmp(const void *v1, const void *v2, size_t n)
```

An den Adressen "v1" und "v2" werden die jeweils ersten "n" Zeichen miteinander verglichen. Diese Funktion ist identisch mit "strncpy", aber sie vergleicht beliebige Daten (allerdings immer byteweise) und bricht nicht beim ersten Nullbyte ab.

### memchr

```
void *memchr(const void *v, int c, size_t n)
```

sucht analog zu "strchr" das Zeichen "c" in den ersten "n" Zeichen des Speichervektor "v" und liefert einen Zeiger auf dieses Zeichen.

### memset

```
void *memset(void *v, int c, size_t n)
```

schreibt das Zeichen "c" in die ersten "n" Bytes ab Adresse "v".

## 1.13 Datum und Uhrzeit: <time.h>

### clock_t

```
typedef unsigned clock_t
```

Der ganzzahlige Datentyp "clock_t" repräsentiert eine Zeitangabe und wird im Zusammenhang mit der Funktion "clock" benutzt.

### time_t

```
typedef unsigned time_t
```

Auch "time_t" ist ein Datentyp und stellt so etwas wie eine komprimierte Form der "DateStamp"-Struktur des Betriebssystems dar.

## tm

```
struct tm
{ int tm_sec,  tm_min,  tm_hour, tm_mday, tm_mon, tm_year,
     tm_wday, tm_yday, tm_idst; }
```

Die Struktur "tm" enthält alles, was zu einer Zeit- und Datumsangabe gehört. Die einzelnen Felder bedeuten dabei:

| | |
|---|---|
| tm_sec | Sekunden (0 bis 61, denn man hat hier lustigerweise sogar daran gedacht, daß es bis zu 2 Schaltsekunden geben kann) |
| tm_min | Minuten (0 bis 59) |
| tm_hour | Volle Stunden seit Mitternacht (0 bis 23) |
| tm_mday | Monatstag (1 bis 31) |
| tm_mon | Monate seit Januar (d. h. von 0 bis 11!) |
| tm_year | Jahre seit 1900, z. B1992 |
| tm_wday | Wochentag: von 0 (Sonntag) bis 6 (Samstag) |
| tm_yday | Tage seit dem 1. Januar (0 bis 365) |
| tm_idst | positiv, wenn Sommerzeit gilt, Null bei Normalzeit und negativ, wenn das unbekannt ist (in MaxonC$^{++}$ immer) |

## time

```
time_t time (time_t *tp)
```

Die Funktion "time" liefert die aktuelle Systemzeit in Form einer "time_t"-Zahl. Wenn ihr Argument nicht Null ist, wird das Ergebnis zusätzlich noch in "*tp" abgelegt.

Normalerweise benutzt man das Ergebnis von "time" mit den Funktionen "difftime", "gmtime" oder "localtime".

## gmtime

```
struct tm *gmtime (const time_t *tp)
```

Der Inhalt von "*tp", also eine Zeitangabe in diesem geheimnisvollen "time_t"-Format, wird in einem ebenso geheimen, internen Puffer in das anwenderfreundliche "tm"-Format umgewandelt. Ein Zeiger auf diesen Puffer wird zurückgegeben, so daß man die Kalender- und Zeitdaten von dort auslesen kann. Es gibt aber nur einen internen Puffer, so daß die Daten beim

nächsten Aufruf von "gmtime" oder ihrer Schwesterfunktion "localtime" wieder überschrieben werden.

## localtime

```
struct tm *localtime(const time_t *tp)
```

Diese Funktion unterscheidet sich von "gmtime" nur darin, daß die Systemzeit in "*tp" hier in Ortszeit umgewandelt wird (daher das "local"), während „gmtime" ein Ergebnis in „Coordinated Universal Time" (UTC) liefern sollte - das "gm" leitet sich übrigens von der alten Bezeichnung „Greenwich Mean Time" ab. Unter MaxonC++ sind die beiden Funktionen aber absolut identisch.

## mktime

```
time_t mktime (struct tm *tp)
```

"mktime" ist das Gegenstück zu "localtime" und wandelt den Inhalt der "tm"-Struktur "*tp" in das komprimierte "time_t"-Format um. Im Prinzip wird dabei verlangt, daß alle, aber auch wirklich alle Einträge der Struktur korrekt initialisiert sind. MaxonC++ ist aber schon zufrieden, wenn die ersten sechs Member (Sekunden, Minuten, Stunden, Tag, Monat, Jahr) mit gültigen Werten initialisiert sind.

## clock

```
clock_t clock (void)
```

Diese Funktion liefert die bisherige Laufzeit des Programms. MaxonC++ implementiert diese Funktion etwas nachlässig, denn "clock" liefert nur die Zeit, die seit dem Programmstart vergangen ist, und nicht die CPU-Zeit, wie es eigentlich verlangt wird. Jedenfalls können Sie das Ergebnis von "clock" in Sekunden umrechnen, indem Sie durch "CLOCKS_PER_SEC" dividieren.

## CLOCKS_PER_SEC

```
#define CLOCKS_PER_SEC 50
```

Diese Konstante gibt den Umrechnungsfaktor zwischen "clock_t" und einer Zeitangabe in Sekunden an. "clock()/CLOCKS_PER_SEC" liefert die bisherige Laufzeit des Programms in Sekunden.

## difftime

```
double difftime(time_t t1, time_t t2)
```

"difftime" subtrahiert die Zeiten "t1-t2" und liefert das Ergebnis als Fließkommazahl in Sekunden.

## strftime

```
int strftime(char *s, unsigned len, const char *format, const struct
tm *t)
```

Diese hochkomfortable Funktion wandelt die Zeit in der Struktur "*t"ähnlich wie die Funktion "sprintf" in eine Stringdarstellung um und legt diese mit einem abschließenden Nullzeichen im Zeichenvektor "s" ab. Dabei werden maximal "len" Zeichen in "s" geschrieben.

"format" ist ein Formatstring, der wie bei "sprintf" nach "s" kopiert wird. Dabei werden Anweisungsfolgen, die jeweils mit einem "%" beginnen, durch ansprechend formatierte aus "*t" ersetzt. Der Rückgabewert ist die Anzahl der Zeichen, die in "s" geschrieben wurden, wobei das abschließende Nullbyte nicht mitgezählt wird. Wenn aber "len" Zeichen nicht reichen, ist das Ergebnis Null.

Die folgende Tabelle gibt die zahlreichen möglichen Umwandlungsoperatoren an:

| | |
|---|---|
| " | Gekürzter Wochentag, z. B. "Mon" für Montag |
| %A | Vollständiger Wochentag, z. B. "Monday" |
| %b | Monats-Kurzname, z. B. "Apr" füt April |
| %B | Voller Name des Monats, z. B. "April" |
| %c | Kurzdarstellung von Datum und Uhrzeit, z. B. "Apr 06 23:55:42 1992" |
| %d | Monatstag ("01" bis "31") |
| %H | Amerikanische Stundendarstellung ("01" bis "12") |
| %I | Stunde auf Europäisch ("00" bis "23") |
| %j | Tag im Jahr ("001" - "366") |
| %m | Monat ("01" - "12") |
| %M | Minute ("00" - „59") |
| %p | "AM" oder "PM" |
| %S | Sekunde ("00" - "61") |
| %U | Woche im Jahr ("00"-"53"), mit Sonntag als ersten Wochentag |
| %w | Wochentag (von "0" für Sonntag bis "6" für Samstag) |
| %W | Woche im Jahr ("00"-"53"), mit Montag als ersten Wochentag |
| %x | Datum, z. B. "Apr 06 1992" |
| %X | Uhrzeit, z. B. "23:55:42" |

`%y` Jahr im Jahrhundert, z. B. `"92"`
`%Y` Vollständige Jahreszahl, z. B. `"1992"`
`%Z` Name der Zeitzone (hier nicht implementiert)
`%%` Prozentzeichen

Beispiel:

```
#include <stream.h>
#include <time.h>

void main()
{
  char s[80];
  time_t t = time(0);

  strftime(s, 80, "Es ist jetzt %X am %d. %m. %Y.\n",
  localtime(&t));
  cout « s;
}
```

gibt so etwas aus wie

**Es ist jetzt 10:47:11 am 07. 04. 1995.**

## asctime

```
char *asctime (const struct tm *t)
```

ist eine Sparversion von `"strftime"` und wandelt die Zeit aus `"*t"` in eine Zeichenkette der Form `"Tue Apr 07 01:03:42 1992"` um. Diese Zeichenkette wird in einem internen Puffer abgelegt und ein Zeiger darauf zurückgegeben.

## ctime

```
char *ctime(const time_t *t)
```

ist identisch mit `"asctime(localtime(t))"`, wandelt also einen `"time_t"`-Wert in eine Stringdarstellung um.

# 2. Bibliotheken für C++

## 2.1 Ein- und Ausgaben in C++: <stream.h> und <iostream.h>

### 2.1.1 Ausgabe

#### 2.1.1.1 „ostream" und der Operator „«"

Bekanntlich ist in C++ alles viel schöner, besser und leichter als in ANSI C, und so ist es auch bei der Ein- und Ausgabe von Daten. Die dazu nötigen Definitionen stehen in der Includedatei „*<stream.h>*", während „*<iostream.h>*" nur so etwas wie ein Aliasname für erstere Datei ist und lediglich ein `"#include <stream.h>"` enthält. Genaugenommen ist „*<stream.h>*" die inzwischen veraltete Bezeichnung (aus dem 1.0-Standard) und „*<iostream.h>*" laut 2.0-Standard korrekter, aber letzten Endes ist das ja egal, denn die beiden Bezeichnungen sind absolut äquivalent. In diesem Handbuch wird konsequent „*<stream.h>*" benutzt.

In diesen Dateien wird unter anderem die Klasse `"ostream"` definiert, die für Ausgaben aller Art zuständig ist. Es gibt drei Standard-Objekte dieses Typs:

`cout` für gewöhnliche Ausgaben
`cerr` für Fehlermeldungen, die sofort ausgegeben werden
`clog` für gepufferte Fehlermeldungen

MaxonC++ nimmt sich die Freiheit, daß `"cerr"` und `"clog"` unterschiedliche Namen für ein und dasselbe Objekt sind.

Auf dieser Klasse `"ostream"` ist vor allem der Operator `"«"` mehrfach überladen. Der linke Operand ist dabei jeweils ein `"ostream"`-Objekt, über das die Ausgabe erfolgen soll, z. B. `"cout"`, und der rechte Operand ist ein Ausdruck eines Standard-Typs. Im einzelnen sind folgende Operatorfunktionen als Member definiert:

**ostream &operator « (int)**
**ostream &operator « (unsigned)**
**ostream &operator « (long)**
**ostream &operator « (unsigned long)**
**ostream &operator « (long long)**
**ostream &operator « (unsigned long long)**
**ostream &operator « (unsigned char)**
**ostream &operator « (signed char)**
**ostream &operator « (char)**
**ostream &operator « (const char *)**
**ostream &operator « (float)**

**ostream &operator « (double)**
**ostream &operator « (long double)**
**ostream &operator « (const void*)**
**ostream &operator « (form &)**
**ostream &operator « (ios&(*f)(ios&))**

Wie unschwer zu erkennen ist, gibt es für jeden numerischen Datentyp eine Ausgabefunktion (die Typen `"short"` und `"unsigned short"` werden gemäß den C++-Typregeln nach `"int"` umgewandelt und mit der entsprechenden Funktion ausgegeben). `"signed char"`- und `"unsigned char"`-Werte werden hier als Zahlen interpretiert, während bei `"char"` ein Zeichen ausgegeben wird. Bei einem Zeichenkettenargument (`"const char*"`) wird natürlich der entsprechende String ausgegeben, während bei einem beliebigen anderen Pointer („const void*") die Speicheradresse als hexadezimale Zahl mit vorangestelltem `"0x"` ausgegeben wird. Das kann natürlich zu Überraschungen führen, wenn man die Adresse einer `"char"`-Variablen ausgeben will:

```
#include <stream.h>

void main()
{
int i;
char c;

cout « &i;          // gibt Adresse von "i" hexadezimal aus
cout « &c;          // verunglückte String-Ausgabe
}
```

Hier versucht der Compiler, der nun einmal hinreichend dämlich ist, `"c"` als String zu interpretieren, und das Programm gibt alle Zeichen von `"c"` bis zum ersten im Speicher folgenden Nullbyte aus.

Außerdem stoßen die beiden letzten Funktionsdeklarationen in der obigen Liste bei Ihnen vielleicht auf Unverständnis. Die Funktion

```
ostream &ostream::operator « (form &)
```

stellt über die Klasse `"form"` eine formatierte Ausgabe im C-Stil zur Verfügung (siehe auch 2.1.1.3), und

```
ostream &ostream::operator « (ios&(*f)(ios&))
```

wendet eine Funktion auf eine Basisklasse von `"ostream"` an (2.1.1.3).

Das Ergebnis all dieser Funktionen ist stets eine Referenz auf das `"ostream"`-Objekt, so daß man mehrere Ausgaben aneinander hängen kann, z. B.:

```
int i=0x2a;
cout « "Der Wert von i ist: " « i « '\n';
```

Auch wenn die Werte natürlich in der Reihenfolge ausgegeben werden, in der man sie hinschreibt, kann ihre Auswertungsreihenfolge durchaus eine andere sein:

```
int i = 40;

cout « ++i « ++i;
```

Die Ausgabe kann hier entweder `"4142"` oder `"4241"` sein (die Ausgabe von Zahlen erfolgt ohne trennende oder führende Leerzeichen), denn voll ausgeschrieben entspricht dieser Ausdruck

```
cout.operator«(++i).operator«(++i)
```

und die Auswertungsreihenfolge in Ausdrücken ist in C++ nur in bestimmten Ausnahmefällen vorgeschrieben.

### 2.1.1.2 Basen und Manipulatoren

Die Klasse `"ostream"` hat noch eine Basisklasse namens `"ios"`. Aus bestimmten Gründen wird `"ostream"` sogar virtuell davon abgeleitet, aber das interessiert momentan nicht so unbedingt. Jedenfalls existieren ein paar Funktionen, nämlich `"dec"`, `"hex"`, `"oct"`, `"endl"` und `"flush"`, die als Parameter eine Referenz auf ein solches `"ios"`-Objekt erwarten und dieselbe Referenz als Ergebnis liefern. Da eine abgeleitete Klasse alle Eigenschaften ihrer Basisklassen erbt, kann man diese Funktionen natürlich auch auf die Klasse `"ostream"` anwenden.

Die drei Funktionen

```
ios &dec (ios &s)
ios &hex (ios &s)
ios &oct (ios &s)
```

setzen die Zahlbasis in einem Strom-Objekt. Nach einem Funktionsaufruf wie

```
hex(cout)
```

werden in Zukunft über `"cout"` alle ganzzahligen Werte hexadezimal, aber ohne vorangestelltes `"0x"`, ausgegeben. Die Ausgabe von Fließkomma- oder Stringwerten wird dadurch nicht beeinflußt.

Analog schaltet die Funktion `"oct"` auf oktale Zahldarstellung um, und `"dec"` stellt wieder den dezimalen Modus her.

Ein Beispiel:

```
#include <stream.h>

void main()
{
```

```
    int i = 42;
    hex(cout);
    cout « "hexadezimal: " « i « "\n";
    dec(cout);
    cout « "dezimal:     " « i « "\n";
    oct(cout);
    cout « "oktal:       " « i « "\n";
    cout « "Es gilt die jeweils zuletzt gewählte Einstellung: "
    « i « "\n";
}
```

Das Programm gibt aus:

**hexadezimal: 2a**
**dezimal: 42**
**oktal: 52**

Es gilt die jeweils zuletzt gewählte Einstellung: 52

Wie Sie an der letzten Ausgabe erkennen können, gelten die so vorgenommenen Einstellungen auch über eine Ausgabezeile hinaus, nämlich so lange, bis wieder ein anderer Modus gewählt wird.

Ein- und Ausgabedateien, und ein C++-Strom repräsentiert im Grunde genommen nichts anderes, werden oft mit einem Puffer beschleunigt (siehe auch 1.2.6.2). Die Funktion

```
ios &flush (ios &s)
```

löscht den Schreibpuffer eines Ausgabestroms, indem sie seinen Inhalt ausgibt. Wenn der Strom keinen Puffer hat, hat die Funktion keinerlei Wirkung. Maxon C++ weist den Standard-Ausgabeströmen als Default zwar keinen Puffer zu, aber auf einigen Systemen ist nach einer Ausgabe ein `"flush(cout)"` bzw. `"flush(clog)"` nötig, um die Ausgabe wirklich zu schreiben.

Natürlich kann man diese Funktion auch so benutzen:

```
flush(cout « "Ausgabe");
```

...oder so:

```
flush(cout) « "Ausgabe";
```

Der erste Ausdruck gibt erst etwas aus und wendet dann `"flush"` an, die zweite Anweisung schreibt erst einen etwaigen Pufferinhalt und macht dann eine weitere Ausgabe, die aber gegebenenfalls natürlich zunächst in einem Puffer aufbewahrt wird.

Last not least gibt es noch die Funktion

```
ios &endl (ios &s)
```

Sie gibt einen Zeilentrenner in einen Strom aus und ruft anschließend `"flush"` auf:

```
cout « "Das Ergebnis ist: " « Resultat;
```

```
endl(cout);
```

oder

```
endl(cout « "Das Ergebnis ist: " « Resultat);
```

entspricht

```
cout « "Das Ergebnis ist: " « Resultat;
flush(cout « "\n");
```

Diese kleinen Funktionen sind zwar ganz praktisch, aber ihre Handhabung und der damit verbundene Schreibaufwand entsprechen, wie auch die Übersichtlichkeit, doch eher dem C-Niveau als einem angenehmen C++-Stil. Deshalb gibt es die Funktion

```
ostream &ostream::operator « (ios&(*f)(ios&))
```

Wenn man die Adresse einer Funktion mit der passenden Signatur in ein `"ostream"`-Objekt „schreibt", wird sie darauf ausgeführt:

```
cerr « hex
```

ist damit äquivalent zu

```
hex(cerr)
```

Das eröffnet dem Programmierer reizvolle Möglichkeiten:

```
cout « hex « 26731 « endl;
```

schaltet die Standard-Ausgabe auf hexadezimale Ausgabe um, schreibt dann in eben diesem Modus eine Zahl und führt `"endl"` aus. Auf diese Weise wird die Handhabung solcher Funktionen doch erheblich angenehmer und intuitiver.

### 2.1.1.3 Formatierte Ausgabe

Die Klasse `"form"` stellt eine formatierte Ausgabe im Stil von `"printf"` zur Verfügung. Sie hat einen Konstruktor, dessen Parameter demgemäß eine nicht ganz unzufällige Ähnlichkeit mit denen von `"printf"` haben:

```
form(const char *fmt, ...)
```

In der Formatzeichenkette können sämtliche Umwandlungsoperationen und Optionen von `"printf"` (siehe 1.2.4) benutzt werden, und an Stelle der Ellipse `"..."` sind die entsprechenden zusätzlichen Argumente zu setzen. Man kann sich ein solches `"form"`-Objekt erzeugen und mit dem gewöhnlichen Operator `"«"` in einen `"ostream"` ausgeben:

```
#include <stream.h>

void main()
{
int i=42;
char s[ ] = "die große Frage";

form f("Die Antwort auf %s ist %d.\n", s, i);

cout « f;
}
```

In dem Objekt `"f"` wird hier der String

```
"Die Antwort auf die große Frage ist 42.\n"
```

abgespeichert und kann beliebig oft ausgegeben werden. Gebräuchlicher ist aber wohl die Möglichkeit, ein namenloses, temporäres Objekt auszugeben:

```
cout « form("Hex: %x", 42) « endl;
```

Auf diese Weise stehen auch bei den C++-Streams sämtliche Formatierungsmöglichkeiten von `"printf"` zur Verfügung. Der Nachteil ist natürlich, daß auch `"form"` eine Funktion ohne Netz und doppelten Boden ist und der Programmierer deshalb selbst darauf zu achten hat, daß die Argumente stimmen.

### 2.1.1.4 Ausgabe binärer Daten

Die Standardausgabe ist in der Regel ein Fenster, oder sie wird auch manchmal in eine (Text-)Datei umgelenkt. Deshalb wird man normalerweise über `"cout"`, `"cerr"` und `"clog"` nur formatierte Textdaten ausgeben. Man kann einen `"ostream"` aber auch an eine Datei anbinden, und dann wird man durchaus manchmal auch binäre Daten schreiben wollen. Deshalb gibt es die beiden Memberfunktionen

```
ostream &ostream::put (char c)
ostream &ostream::write (const char *p, int n)
```

Erstere Funktion schreibt ein Zeichen in einen Ausgabestrom, letztere die ersten `"n"` Bytes ab Adresse `"p"`. Daß `"p"` hier ein Zeiger auf `"char"` ist, sollte Sie nicht weiter beunruhigen, denn natürlich können Sie beliebige andere Zeiger nach `"char*"` casten.

Einige Beispiele:

```
cout.write("Hello, World!",5)
```

schreibt die Zeichenfolge `"Hello"` nach `"cout"`, und

```
double d = 3.14159;
cout.write((char*) &d, sizeof(double));
```

schreibt die `"double"`-Zahl `"d"` in denselben Strom, aber nicht als lesbare, formatierte Zahldarstellung, sondern in Form von (hier)acht Bytes.

Das erste Beispiel ist durchaus sinnvoll (manchmal möchte man wirklich nur einen Teil eines Strings ausgeben, oder man hat eine Zeichenkette vorliegen, bei der das Nullbyte am Ende fehlt), während das letztere wirklich nur bei Dateioperationen einen sittlichen Nährwert hat.

## 2.1.2 Eingabe

### 2.1.2.1 Texteingaben mit „»“

Was Krupp in Essen, wir im Trinken und `"ostream"` bei der Ausgabe, ist die Klasse `"istream"` für die Eingabe von Daten. Es gibt ein Standard-Objekt dieser Klasse namens `"cin"`, das in „*<stream.h>*“ deklariert wird und die Standard-Eingabe eines Programms darstellt, also normalerweise die Eingaben des Benutzers.

Die folgenden Operatoren sind als Member von `"istream"` deklariert:

**istream & operator » (char s[ ])**
**istream & operator » (char &c)**
**istream & operator » (signed char &sc)**
**istream & operator » (unsigned char &uc)**
**istream & operator » (short &s)**
**istream & operator » (unsigned short &us)**
**istream & operator » (int &i)**
**istream & operator » (unsigned &u)**
**istream & operator » (long &l)**
**istream & operator » (unsigned long &ul)**
**istream & operator » (long long &vl)**
**istream & operator » (unsigned long long &uvl)**
**istream & operator » (float &f)**
**istream & operator » (double &d)**
**istream & operator » (long double &ld)**

Linker Operand ist jeweils ein Objekt der Klasse `"istream"`, also insbesondere die Variable `"cin"`, und eine Referenz auf genau dieses Objekt ist auch der Rückgabewert dieser Operatorfunktionen. Als rechter Operand kann, wie Sie der imposanten Auflistung von Funktionen unschwer entnehmen können, ein L-Wert eines beliebigen numerischen Standardtyps (einschließlich `"char"`) oder eine Stringvariable dienen. Die Operatorfunktion liest dann die benötigte Anzahl von Zeichen aus dem Eingabestrom, wandelt sie in den gewünschten Datentypen um (z. B. eine Ziffernfolge in eine ganze Zahl) und weist das Resultat dem rechten Operanden zu.

Ein Beispiel:

```
int i;
char c;
char s[50];

cin » c » i » s;
```

Bei der Eingabe

```
0815-4711 42
```

wird "c" das Zeichen "0" zugewiesen, "i" bekommt den Wert "815" und in "s" wird die Zeichenkette "-4711" abgelegt, während die Zeichen "42" vorerst unbeachtet bleiben und vielleicht bei der nächsten Eingabe Beachtung finden. Liest man auf diese Weise ein "char", werden zuerst Leerzeichen, Zeilentrenner und sonstige Leerraum-Zeichen überlesen. Beim Lesen von Strings werden ebenfalls Trennzeichen überlesen, und es wird bis ausschließlich zum nächsten nachfolgenden Trennzeichen gelesen. Somit wirkt also - zumindest beim Operator "»" - auch ein Leerzeichen als Trennung zwischen zwei Strings.

Auch die Klasse "istream" kann nicht hellsehen, und deshalb weiß die Eingabe-Operatorfunktion "»", wenn sie auf eine Stringvariable angewandt wird, nicht, wie lang dieser String ist. Das kann zu üblen Abstürzen führen, wenn der Anwender bei

```
char input[20];

cin » input;
```

mehr als 20 Zeichen eingibt. Um dies halbwegs sicher zu gestalten, liest die Funktion in Maxon C++ maximal "STREAM_MAXSTRING" Zeichen in Stringvariablen, und zwar einschließlich des Nullbytes am Ende. Diese Konstante wird in „*<stream.h>*" als 80 definiert - ein kanonischer Wert für eine vernünftige Zeilenlänge. Leider handelt es sich dabei um eine Spezialität von MaxonC++ und nicht um einen wirklichen Standard.

Ganzzahlige Werte können wahlweise dezimal, oktal (mit führender Null) oder hexadezimal (mit "0x" oder "0X" am Anfang) eingegeben werden. Fließkommazahlen werden natürlich nur dezimal akzeptiert und können einen Exponententeil (mit "e" oder "E" eingeleitet) haben.

### 2.1.2.2 Eingabefunktionen auf „istream"

Die Memberfunktion "read" liest aus einem "istream" eine beliebige Anzahl von Bytes und legt sie in einem Zielvektor ab. Damit ist sie das Gegenstück zur ANSI-Funktion "fread". Sie ist definiert als

```
istream & istream::read(char buf, int n)
```

Auch hier sollte es Sie nicht weiter beunruhigen, daß der Zielvektor "buf" formal ein "char"-Array ist, denn man kann natürlich jeden Pointer in ein "char*" umwandeln. "n" ist die Anzahl

der Bytes, die gelesen werden sollen, und muß natürlich positiv sein. Das Ergebnis der Funktion ist wie immer eine Referenz auf das `"istream"`-Objekt.

Drei weitere Funktionen, die allesamt `"get"` heißen, lesen jeweils ein Zeichen aus dem Eingabestrom und sind folgendermaßen definiert:

```
istream & istream::get (char &c)
istream & istream::get (unsigned char &c)
int  istream::get ()
```

Die beiden ersten Funktionen lesen das Zeichen in eine `"char"`-Variable und liefern die übliche Referenz als Ergebnis. Sie sind gleichermaßen auf `"char"` und `"unsigned char"` definiert. Die dritte Funktion expandiert das Zeichen nach `"int"` und liefert es als Ergebnis.

Der Unterschied zwischen

```
cin » c
```

und

```
cin.get(c)
```

besteht darin, daß `"get"` keinen Leerraum überspringt, also zeichenweise die vollständige Eingabe einschließlich aller Leerzeichen und Zeilentrenner liest.

Oft merkt man, daß man genug gelesen hat, erst, nachdem man schon zu viel gelesen hat. In diesem Fall hilft die Funktion `"putback"`, die ein bereits gelesenes Zeichen in den Eingabestrom zurückstellt:

```
istream & istream::putback(char c)
```

Auf diese Weise kann aber immer höchstens ein Zeichen gepuffert werden.

Besonders beim Lesen aus Dateien muß man irgendwie wissen, wann man am Ende der Datei angelangt ist. Deshalb bietet `"istream"` auch eine Memberfunktion, die eben dieses testet:

```
int istream::eof()
```

Das Ergebnis dieser Funktion ist von Null verschieden, wenn das Dateiende überschritten wurde. Damit entspricht die Funktion `"feof"` aus „*<stdio.h>*“.

Nun fehlt noch eine Funktion, die eine Textzeile liest. Hier ist sie:

```
istream &getline(char *buf, int len, char Delim = '\n')
```

Es werden maximal `"len"`-1 Zeichen in den Vektor `"buf"` gelesen. Die Funktion bricht ab, sobald ein Trennzeichen `"Delim"` gelesen wird - als Default ist dies der Zeilentrenner. Dieses Trennzeichen wird zwar aus dem Eingabestrom gelesen und nicht wieder zurückgestellt, aber nicht in `"buf"` abgelegt.

## 2.1.3 Weitere Funktionen

### 2.1.3.1 Die Schnittstelle zum ANSI C-Dateisystem

Oben wurde bereits angedeutet, daß die `"stream"`-Funktionen von MaxonC++ im wesentlichen verkleidete ANSI-C-Funktionen sind. Diese Schnittstelle kann der Programmierer durch die Funktion `"getstream"` der Klasse `"ios"` nutzen:

```
FILE * ios::getstream()
```

Diese Funktion liefert als Ergebnis einen Zeiger auf den ANSI C- Filedeskriptor. Damit lassen sich sämtliche Funktionen aus „*<stdio.h>*" auch auf C++-Streams benutzen.

Außer den hinlänglich bekannten Objekten `"cout"`, `"cin"` und so weiter kann man auch selbst Objekte der Klassen `"istream"` und `"ostream"` erzeugen. Zu diesem Zweck haben die beiden genannten Klassen jeweils einen Konstruktor, der als Argument einen Dateizeiger erwartet.

Ein Beispiel:

```
#include <stream.h>
#include <stdio.h>

void main()
{
   FILE *f = fopen("test", "r");                    // C-File öffnen
   if (f)
   {
      istream is(f);                                // eigenes "istream"-
Objekt
      char s[80];

      is » s;                                       // liest einen String

      fclose(f);                                    // Datei wieder schließen
   }
}
```

`"istream"` und `"ostream"` haben keine Destruktoren, die die Dateien nachher automatisch schließen. Schon allein deshalb ist es nicht unbedingt sinnvoll, auf diese Weise mit Dateien zu arbeiten - mit den Klassen und Funktionen aus „*<fstream.h>*" geht das viel einfacher.

### 2.1.3.2 Fehlererkennung

Mit einer etwas eigenartigen Funktion bieten die Strom-Klassen eine äußerst intuitive Möglichkeit, Fehler in Dateien zu erkennen: In der Basisklasse `"ios"` gibt es eine Konvertierungsfunktion

```
ios::operator void*()
```

Diese Funktion liefert `"0"`, wenn das Dateiende überschritten oder ein anderer Fehler bemerkt wurde, und andernfalls einen Zeiger auf das Objekt selbst. Das sieht zunächst seltsam aus, ist aber

um so einfacher anzuwenden: Damit kann man Objekte und Ausdrücke des Typs `"ios"` oder davon abgeleiteter Klassen direkt in Abfragen einsetzen, z. B. so:

```
cin » s;
if (cin)
{ usw.
```

...oder auch so:

```
 char s[80];

while (cin » s)
cout « s;
```

Solche Ausdrücke sind logisch wahr, wenn kein Fehler aufgetreten und das Dateiende nicht erreicht ist. Natürlich kann man auf solche Abfragen auch die logischen Operatoren `"!"`, `"&&"` und `"||"` anwenden.

Kanonisch wäre natürlich gewesen, statt dieses komischen `"void"`-Zeigers eine Konvertierungsfunktion nach `"int"` oder einem anderen numerischen Typ zu implementieren, denn ein Zeiger als Bedingung ist doch ein wenig gewöhnungsbedürftig. Das ist aber nicht möglich, weil die Standard-Operatoren `"»"` und `"«"` auf ganzzahligen Werten definiert sind und Ausdrücke wie `"cout « n"` damit mehrdeutig würden.

## 2.2 Ströme und Dateien: <fstream.h>

### 2.2.1 Klassen für Dateien

Die Standard-Ein- und Ausgabeströme sind zwar ganz nett, aber natürlich braucht man daneben auch „richtige" Dateien, und es wäre angenehm, wenn man alle Features aus „*<stream.h>*" auch mit beliebigen Dateien benutzen könnte. Deshalb gibt es die Includedatei „*<fstream.h>*", in der Klassen und Funktionen für Dateien definiert werden.

In *<fstream.h>* werden drei für Sie interessante Klassen definiert:

- `"ifstream"` für Eingaben aus Dateien,
- `"ofstream"` für Ausgaben in Dateien,
- `"fstream"` für Ein- und Ausgaben.

`"ifstream"` ist von `"istream"` (und damit indirekt von `"ios"`) abgeleitet und erbt damit alle in 2.1.2 vorgestellten Eigenschaften. Entsprechend wurde `"ofstream"` von `"ostream"` abgeleitet, während `"fstream"` - ein interessantes Beispiel für Mehrfachvererbung - gleichzeitig `"istream"` und `"ostream"` als Basisklassen hat.

Die drei neuen Klassen haben außer diesen ererbten Funktionen auch noch einen Satz von elementaren Dateifunktionen, etwa Öffnen und Schließen von Dateien. Diese gemeinsamen Eigenschaften werden in der Klasse `"fstreambase"` zusammengefaßt, die ebenfalls Basisklasse von `"ifstream"`, `"ofstream"` und `"fstream"` ist.

## 2.2.2 Dateien öffnen und schließen

Die drei oben vorgestellten Datei-Stromklassen besitzen jeweils einen Default-Konstruktor. Wird aber ein Klassenobjekt damit erzeugt, ist es zunächst an keine physikalische Datei „angebunden", und alle Ein- und Ausgabeoperationen führen unabwendbar zu einem Absturz. Deshalb muß zunächst die Member-Funktion `"open"` der gemeinsamen Basisklasse `"fstreambase"` aufgerufen werden:

```
FILE * fstreambase::open (const char *name, int mode)
```

Der erste Parameter steht, wie man sich denken kann, für den Dateinamen, während der zweite den Zugriffsmodus festlegt. Mögliche Werte dafür werden von einem Aufzählungstyp, der in der Klasse `"ios"` verborgen ist, definiert:

```
enum open_mode
{
in  = 1,          // Eingaben
out = 2,          // Ausgaben
app = 4           // Anhängen
}
```

Beim Modus `"out"` wird stets eine neue Datei erzeugt. Existiert bereits eine Datei gleichen Namens, so wird diese gelöscht.

Das folgende Programm öffnet eine Datei und schreibt eine Zeile hinein:

```
#include <fstream.h>

const char dateiname[ ] = "ram:Test";

void main()
{
ofstream os;
os.open(dateiname, ios::out);
os « "Dies ist die Datei " « dateiname « ".\n";
os.close();
}
```

Wie Sie sehen, haben wir oben bereits eine weitere Funktion aus der Klasse `"fstreambase"` benutzt, nämlich `"close"`. Diese Funktion schließt die Datei, an die der Strom gebunden ist. Ein überflüssiges `"close"` schadet nicht. Nach dem Schließen ist das Stream-Objekt wieder an keinerlei Datei angekoppelt, und man kann, wenn man will, erneut eine Datei öffnen.

Kommen wir noch einmal auf die Zugriffsmodi von `"open"` zurück: Die drei vorgegebenen Werte können durch bitweises „Oder" miteinander kombiniert werden. Beispielsweise steht

```
ios::in|ios::out
```

für Lesen und Schreiben zugleich, oder beim Modus

```
ios::out|ios::app
```

wird zwar aus der Datei gelesen, aber der Lesezeiger steht zunächst am Ende der Datei und muß dann z. B. mit `"seekg"` (siehe unten) zurückgesetzt werden.

## 2.2.3 Fehlererkennung

Selbstverständlich sollte man immer, wenn man eine Datei öffnen läßt, prüfen, ob das auch wirklich geklappt hat, bevor man etwas mit der Datei anstellt. Dazu dient wieder die normale Fehlerkontrolle mit der eigenartigen Konvertierungsfunktion aus 2.1.3.2. Diese ist nämlich immer logisch „falsch", wenn ein Strom an keine Datei angekoppelt ist.

Das folgende Programm demonstriert dies, indem es eine Datei zum Lesen öffnet und sie dann in eine andere Datei kopiert:

```
#include <fstream.h>

void main()
{
   ifstream is;
   is.open("s:startup-sequence", ios::in);

   if(is)
   {
      ofstream os;
      os.open("ram:copy_of_s-seq", ios::out);

      if (os)
      {
         char s[80];
         while (is.getline(s,80))
            os « s « "\n";
         os.close();
      }
      else
         cout « "Kann Ausgabedatei nicht öffnen";

      is.close();
   }
   else
      cout « "Kann Eingabedatei nicht öffnen";
}
```

## 2.2.4 Konstruktoren und Destruktoren

Natürlich geht in C++ alles viel einfacher als in C, und so ist es auch bei diesen Dateioperationen. Ich verschwieg Ihnen bisher, daß `"ifstream"`, `"ofstream"` und `"fstream"` jeweils nicht nur einen Default-Konstruktor haben, sondern auch einen solchen, dessen Parameter mit denen der `"open"`-Funktion identisch sind. Dadurch kann man die Definition eines Dateiobjekts mit dem Öffnen der Datei aufs einfachste kombinieren, z. B. so:

```
ofstream outf("C++:Ausgabe", ios::out);
```

Außerdem ist für diesen Konstruktor bei `"ofstream"` als Default-Argument für den zweiten Parameter `"ios::out"` definiert, so daß die Deklaration sich verkürzt zu:

```
ofstream outf("C++:Ausgabe");
```

Einen entsprechenden Konstruktor gibt es auch für die Klasse `"ifstream"`, nur natürlich mit `"ios::in"` als Default-Argument. Auch für `"fstream"` existiert ein solcher Konstruktor, hier aber ohne einen abkürzenden Default-Modus.

Auch nachdem man ein Objekt auf diese erzeugt hat, muß man natürlich zuerst prüfen, ob die Datei auch wirklich geöffnet werden konnte, bevor man irgendwelche Ein- oder Ausgaben tätigt.

Außerdem haben die drei File-Stream-Klassen jeweils einen Destruktor, der die an das Objekt angebundene Datei schließt, sofern dies nicht schon geschehen ist. Er macht das natürlich unabhängig davon, mit welchem Konstruktor das Objekt erzeugt wurde.

Als Beispiel folgt noch einmal das in 2.2.3 vorgestellte Programm, diesmal aber unter Ausnutzung der Kon- und Destruktoren und mit vereinfachter Fehlerbehandlung:

```
#include <fstream.h>

void main()
{ ifstream is("s:startup-sequence", ios::in);
  ofstream os("ram:another_copy_of_s-seq", ios::out);

  if(is && os)
  { char s[80];
    while (is.getline(s,80))
      os « s « "\n";
    os.close();
  }
  else
    cout « "Kann nicht beide Dateien öffnen.";
}
```

## 2.2.5 Puffer

Auch bei den Strömen gibt es wieder das alte Elend, daß Dateioperationen nervtötend langsam sind, wenn byteweise oder in ähnlich kleinen Happen gelesen oder geschrieben werden muß. Deshalb gibt es in der Klasse `"fstreambase"` (also der gemeinsamen Basisklasse der drei Stromklassen) zwei Funktionen, deren Funktion sich dem geneigten Leser möglicherweise intuitiv erschließt:

```
int fstreambase::buffer (unsigned s)
void fstreambase::flush ()
```

`"buffer"` verleiht einer Datei einen Puffer der Größe `"s"` und entspricht damit einem „*<stdio.h>*"-Aufruf wie

```
setvbuf(getstream(), 0, _IOFBF, s)
```

Man kann einen Puffer für eine Datei immer nur einmal setzen und danach seine Größe nicht mehr verändern. Der benötigte Speicherplatz wird automatisch angefordert und beim Schließen der Datei ebenso automatisch wieder freigegeben. Das Ergebnis von `"buffer"` ist übrigens Null, wenn dabei kein Fehler auftrat.

`"flush"` schreibt den Inhalt eines Schreibpuffers in die Datei.

## 2.2.6 Positionieren in Strömen

Von den grundlegenden Dateioperationen fehlen jetzt nur noch Operationen zum Ermitteln und Setzen der Dateiposition. Zunächst wäre da die Funktion

```
int fstreambase::seekg (long offset, int mode = ios::beg)
```

Hier gibt es drei verschiedene Modi, die in Form eines Aufzählungstyps versteckt in `"ios"` definiert werden:

```
enum seek_dir
{
  beg = -1, // absolute Position zum Dateianfang
  cur = 0,      // relative Verschiebung von der akt. Position
  end = 1   // (negativer) Offset zum Dateiende
}
```

Das folgende Programm macht dem Rechner ein `"x"` für ein `"u"` vor, indem es in einer Datei alle `"u"` und `"U"` durch `"x"` bzw. `"X"` ersetzt. Dazu braucht man natürlich ein `"fstream"`-Objekt, denn man muß ja gleichermaßen lesen (um die „U"s zu suchen) und schreiben (nämlich die `"X"`-e). Die Datei wird hier zeichenweise eingelesen, und immer, wenn ein `"U"` gefunden wurde, wird die Position mit `"seekg"` um ein Zeichen zurückgesetzt und dann eben dieses `"U"` überschrieben:

```
#include <fstream.h>

const char name[ ] = "ram:test";

void main()
{
  fstream f(name, ios::in|ios::out);

  while (f)
  {
    char c = f.get();                    // Zeichen lesen

    if (c == 'u' || c == 'U')            // "U" ?
    {
      f.seekg(-1, ios::cur);             // dann zurück
      f.put( c=='u' ? 'x' : 'X' );       // und "X" schreiben
    }
  }
}
```

Als letztes lernen Sie jetzt noch eine Funktion kennen, mit der Sie die aktuelle Dateiposition ermitteln können. Sie heißt `"tellg"`:

```
long fstreambase::tellg ()
```

und liefert, wenn sie auf einem `"ifstream"`-, `"ofstream"`- oder `"fstream"`-Objekt aufgerufen wird, die Position bezüglich des Dateianfangs in Bytes. Die erste Position hat wie immer die Nummer Null.

## 2.3 Ein paar wichtige Definitionen in <new.h>

In der Datei „*<new.h>*" steht eigentlich fast nichts: Zum einen wäre da der Datentyp `"size_t"`, der mit `"unsigned"` identisch ist und auch in diversen anderen Files definiert wird, außerdem die hinlänglich bekannte Konstante `"NULL"` und die folgende Funktion:

```
void (*set_new_handler(void(*)(void))) (void)
```

Man muß natürlich dreimal hinsehen, um diese Deklaration als Funktion zu erkennen. Einfacher wird's, wenn man die Deklaration mit einem `"typedef"` aufteilt:

```
typedef void (*ftype) (void);
ftype set_new_handler (ftype);
```

Also erwartet die Funktion `"set_new_handler"` als Argument einen Zeiger auf eine Funktion, die weder Parameter noch Ergebnistyp hat, und gibt auch einen solchen zurück. Sinn und Zweck dieser Funktion ist es, eine Handler-Funktion zu setzen, die Speichermangel bei `"new"` oder `"malloc"` handhabt.

Normalerweise ignoriert es die Laufzeitbibliothek, wenn bei `"new"` oder `"malloc"` kein Speicher angefordert werden konnte, und liefert dann lediglich einen Nullzeiger, mit dem das Programm dann gefälligst selbst klarkommen muß. Rein praktisch bedeutet das, daß hinter jedes `"new"`, `"malloc"`, `"calloc"` usw. so etwas wie eine `"if"`-Abfrage gehört, die eine solche fehlgeschlagene Speicheranforderung abfängt und das Programm dann irgendwie sinnvoll weiterführt oder sauber beendet.

Bequemer kann man dieses lösen, indem man mit `"set_new_handler"` eine Funktion setzt, die in Zukunft immer dann aufgerufen wird, wenn der Speicher nicht ausreicht. Nach Ausführung dieser Funktion (sofern dort nicht mit `"exit"` o. ä. ausgestiegen wird) versucht die Bibliothek noch einmal, den benötigten Speicher anzufordern, und ruft im Mißerfolgsfall erneut den Handler auf.

Wählt man einen Nullzeiger als Argument für `"set_new_handler"`, ist wieder die Standard-Einstellung aktiv, d. h. das System kümmert sich nicht weiter um den Speicherüberlauf. Das Ergebnis von `"set_new_handler"` ist in jedem Fall die bisher gültige Einstellung, also ein Zeiger auf eine zuvor gesetzte Handlerfunktion oder eben Null.

Die Funktion `"set_new_handler"` wird auch in Abschnitt 2.6.6 des C++-Tutorials erläutert. Das folgende Beispielprogramm fordert gnadenlos so lange Speicherblöcke an, bis der Handler `"coredumped"` aufgerufen wird.

```
#include <new.h>
#include <stream.h>

unsigned MemUsed = 0;

void coredumped()
{
   cout « "Speicherüberlauf nach " « MemUsed « " Bytes.\n";
   cout « "Programm steigt aus!!!\n";
   exit(42);
}

void main()
{
   const unsigned int size = 10000;

   set_new_handler(coredumped);

   for(;;)
   {
      char *cp = new char[size];
      MemUsed += size;
   }
}
```

# __MEMFLAGS

```
extern unsigned int __MEMFLAGS;
```

Beim Operator **"new"** und in den C-Funktionen **"malloc"**, **"calloc"** usw. wird stets beliebiger Speicher alloziert und mit Nullbytes gefüllt. Dieses Verhalten kann man aber ändern: Die Variable **"__MEMFLAGS"** enthält die Flags, die von der Standardbibliothek an die Exec-Funktion **"AllocMem"** übergeben werden. Standardmäßig ist diese Variable mit dem Wert **0x10000**, also **"MEMF_CLEAR"**, initialisiert.

Wenn Sie mit MaxonC++ ein Programm entwickeln, daß später auch auf anderen Computern und Compilern laufen soll, sollten Sie unbedingt am Anfang ihres Programms **"MEMF_CLEAR"** auf Null setzen, um so die Initialisierung des reservierten Speichers mit Nullbytes zu unterdrücken, und vielleicht noch zusätzlich ein Tool wie *„MungWall"* verwenden. Dann können Sie sicher sein, daß diese kleine Inkompatiblität Ihnen beim Portieren keine Probleme bereiten wird.

Eine andere denkbare Anwendung von **"__MEMFLAGS"**: Wenn Sie hier eines der Flags wie **"MEMF_CHIP"** oder **"MEMF_PUBLIC"** eintragen, können Sie die Speicherklasse der nachfolgenden **"new"**- oder „**malloc**"-Aufrufe festlegen.

# 2.4 Ausnahmebehandlung: <exception.h>

## class Exception

```
class Exception
 { public:
     virtual ~Exception() { }
 };
```

Wie im Tutorial in Kapitel 8.3 schon angedeutet, greift die Klasse `"Exception"` dem sehnlichst erwarteten ANSI C++-Standard vor und definiert eine Basisklasse für alle Ausnahmen. Wenn Sie Klassen für Ihre Ausnahmen definieren, können Sie die von dieser Klasse ableiten - müssen Sie aber nicht.

## unexpected

```
void unexpected();
```

Diese Funktion wird aufgerufen, wenn in erier Funktion it Ausnahme-Deklaration eine unerwartete Exception geworfen wird (siehe auch Tutorial 8.6). Normalerweise wird man sie nicht „von Hand" aufrufen.

`"unexpected"` ruft die zuletzt mit `"set_unexpected"` gesetzte Funktion auf. Default ist `"terminate"`.

## terminate

```
void terminate();
```

`"terminate"` wird normalerweise nur von `"unexpected"` aufgerufen und enthält selbst nur einen Sprung an die zuletzt mit `"set_terminate"` gesetzte Funktion, was defaultmäßig `"abort"` ist.

## set_unexpected

```
Definiton: void (*set_unexpected(void(*)()))();
```

Wie oben bereits erwähnt, kann man mit `"set_unexpected"` festlegen, in welche Funktion `"unexpected"` einspringen soll. Als Ergebnis wird die bisher gültige Funktion zurückgegeben.

### set_terminate

```
void (*set_terminate(void(*)()))();
```

Analog zu `"set_unexpected"` verändert diese Funktion das Verhalten von `"terminate"`. Lesen Sie dazu auch das Kapitel 8.6 im Tutorial.

## 2.5 Eine String-Library: <tools/str.h>

### *2.5.1 Die Klasse „String"*

Im Unterverzeichnis *„tools"*, das für Hilfsbibliotheken aller Art vorgesehen ist, finden Sie eine Datei *„str.h"*, die Ihnen eine C++ angemessene Stringverwaltung beschert.

Im wesentlichen wird dort eine Klasse namens `"String"` deklariert, die dynamische Zeichenketten repräsentiert. Das bedeutet: Man deklariert lediglich eine Variable des Typs `"String"`, und die Bibliothek sorgt selbst dafür, daß stets genug - und, einmal abgsehen von einem gewissen Overhead, auch nie zuviel - Speicher dafür reserviert wird. Im Tutorial wird in Kapitel 4.2.3 eine ähnliche (aber weniger leistungsfähige) Library als Beispielprogramm vorgestellt und implementiert.

Die Klasse `"String"` besitzt vier verschiedene Konstruktoren:

```
String (const char *s = 0)
String (int l, char *s)
String (const String &S)
String (char c)
```

Der erste Konstruktor initialisiert einen String mit einer Zeichenkette (oder dem Leerstring als Default) und dürfte der bei Variablendeklarationen meistgebrauchte sein, z. B.

```
String s1, s2 = "Horrido!", s3 = 0, s4 = "";
```

Hier werden die Variablen `"s1"`, `"s3"` und `"s4"` jeweils als Leerstrings initialisiert, während in `"s2"` die Zeichenkette `"Horrido!"` abgespeichert wird. Dazu muß der Konstruktor natürlich Speicher anfordern und dies in `"s2"` vermerken. Um solchen Speicher wieder freizugeben, wenn die Stringvariable nicht mehr existiert, besitzt die Klasse `"String"` auch einen Destruktor.

Den zweite Konstruktor benötigt man hauptsächlich, wenn man der Bibliothek eigene Stringfunktionen hinzufügen will. Aus einem `"char[ ]"`-Objekt, das mit `"new"` erzeugt worden sein muß und mit einem String, der ausschließlich des abschließenden Nullbytes die Länge `"1"` hat, initialisiert sein muß, wird hier ein `"String"`-Objekt erzeugt. Dabei wird die Zeichenkette `"s"` nicht noch einmal kopiert, sondern der Zeiger darauf direkt in das Stringobjekt eingetragen.

Der dritte Konstruktor ist, wie man sieht, ein Copy-Konstruktor, und der letzte initialisiert das Objekt mit einem Zeichen, das als Zeichenkette der Länge `"1"` interpretiert wird. Damit sind

```
String s1 = "?"
```

und

```
String s1 = '?'
```

mehr oder weniger äquivalent.

## 2.5.2 Member-Funktionen

Mit den oben aufgeführten Konstruktoren gibt es eine Konvertierung von den ordinären Zeichenvektoren, mit denen sich der C-Programmierer herumschlagen muß, in die Stringklasse. Für Kompatibilität in die andere Richtung sorgt die Konvertierungsfunktion

```
String::operator char*() const
```

Sie wandelt ein `"String"`-Objekt in einen Zeiger auf eine Zeichekette um. Dabei sollte man sich aber hüten, diese Konvertierung auf temporäre Objekte anzuwenden, z. B. auf das Ergebnis der unten vorgestellten Operatorfunktion `"+"`, denn natürlich liefert sie lediglich einen Zeiger auf den ansonsten „geheimen" Speicherbereich, in dem der eigentliche Wert des Stringobjekts abgespeichert wird, und wenn die Stringvariable nicht mehr existiert oder einen neuen Wert zugewiesen bekommt, muß man stets damit rechnen, daß dieser Speicherbereich freigegeben und wieder überschrieben wird.

Es gibt zwei Zuweisungsoperatoren:

```
String & String::operator = (const String &S)
String & String::operator = (const char *s)
```

Die erste Funktion funktioniert, wie Sie sich denken können, analog zum Kopier-Konstruktor, gibt aber den Speicher, der für den bisherigen Variablenwert alloziert wurde, wieder frei. Die zweite ist eigentlich redundant, denn der Compiler ist natürlich durchaus clever genug, bei einer Wertzuweisung wie

```
String s1;
char c[ ] = "Frood";

s1 = c;
```

aus dem Zeichenvektor mit Hilfe des entsprechenden Konstruktors ein temporäres Objekt zu erzeugen und dieses dann mit der erstgenannten `"="`-Funktion in das Zielobjekt zu kopieren. Mit dieser zusätzlichen Funktion wird dies aber deutlich abgekürzt.

Ungefähr wie "strcat" in Standard-C, nur eben erheblich bequemer, funktioniert die Member-Funktion

```
String & String::operator += (const String &s),
```

die zwei Strings aneinander hängt: Nach

```
String s1 = "Moin ", s2 = "Moin!";

s1 += s2;
```

hat "s1" den Wert "Moin Moin!".

Sie wollen auf einen String zeichenweise zugreifen? Null Problemo, auch das geht, und zwar mit der folgenden Operator-Funktion:

```
char String::operator[ ] (int i) const
```

Sie liefert als Ergebnis das "i"-te Zeichen einer Stringvariablen, allerdings nicht als Referenz und damit auch nicht als L-Wert. Deshalb kann man mit Indizierung Zeichen aus einem String auslesen, aber keine einzelnen Zeichen verändern. Dies wurde aus Sicherheitsgründen so eingerichtet.

Um die Länge eines Strings festzustellen, können Sie die Standard-Funktion "strlen" benutzen, denn es existiert ja eine Konvertierung von "String" nach "char*". Angenehmer und schneller ist aber die folgende Member-Funktion:

```
int String::length() const
```

Sie liefert die Länge eines Strings, wobei das Nullbyte am Ende wie immer nicht mitgezählt wird. Aufgerufen wird sie natürlich in der Form

```
i = s1.length();
```

Zu guter Letzt wären da noch drei Funktionen, die irgendwie an alte BASIC-Tage erinnern:

```
String String::left (int n) const
String String::right (int n) const
String String::mid (int pos, int n) const
```

"left" liefert die ersten "n" Zeichen eines Strings, "right" die letzten "n". Falls der String kürzer als "n" ist, ist das Ergebnis mit dem Argument identisch. "mid" nimmt aus einem String "n" Zeichen ab der Position "pos", wobei letztere wie immer von Null an gezählt wird.

## 2.5.3 Der Operator „+" auf Strings

Das waren alle Funktionen, die als Member von `"String"` deklariert sind. Daneben gibt es noch einige global deklarierte Operatorfunktionen:

Die angenehmste ist sicherlich

```
String operator + (String s1, String s2)
```

Dieses `"+"` hängt, wie man es auch nicht anders erwartet, zwei Strings aneinander. Anders als bei `"strcat"` oder dem Operator `"+="` werden hier die Werte der Argumente keineswegs verändert, sondern es wird ein neues (temporäres) Objekt erzeugt, falls das Ergebnis nicht sogar direkt in das gewünschte Ziel geschrieben wird.

Da es sonst keine Addition auf Standard-Strings (d. h. char-Zeigern) gibt, kann dieser neue Plus-Operator auch dort angewendet werden:

```
#include <tools/str.h>
#include <stream.h>

void main()
{  char *s1 = "Grunz",
   *s2 = s1 + "wanzling";

   cout « s2;
}
```

Eigentlich ist dieses Beispiel korrekt, denn bei der Addition wird der `"+"`-Operator aus der String-Bibliothek benutzt und dessen Ergebnis, ein temporäres Objekt der Klasse `"String"`, mit der Konvertierungsfunktion nach `"char*"` gewandelt. Dabei wird aber der alte Fehler begangen, einen Zeiger auf ein temporäres Objekt zu verwenden. Wenn `"s2"` ausgegeben wird, existiert das temporäre Objekt (und mit ihm sein Zeichenvektor) schon lange nicht mehr, und die Ausgabe ist undefiniert.

So hätte es natürlich geklappt:

```
#include <tools/str.h>
#include <stream.h>

void main()
{  char *s1 = "Grunz";
   String s2 = s1 + "wanzling";

   cout « s2;      // Ausgabe für "String" existiert - siehe unten
}
```

Hier wird - je nach Compiler - die String-Summe entweder gleich im Zielobjekt `"s2"` konstruiert, oder das hier eingeführte temporäre Objekt wird mit dem Copy-Konstruktor in `"s2"` übertragen, bevor es destruiert wird. Also funktioniert `"+"` hier so, wie man es erwartet.

Deshalb soll hier noch einmal erwähnt werden, daß die Konvertierungsfunktion von `"String"` nach `"char*"` nur mit Vorsichtig und Bedacht zu benutzen ist, denn andernfalls erlebt man Überraschungen.

## 2.5.4 Vergleiche

Natürlich kann man String-Objekte mit der Funktion `"strcmp"` vergleichen. Mit den folgenden Operatoren geht das aber wesentlich bequemer:

```
int operator == (const String &s1, const String &s2)
int operator != (const String &s1, const String &s2)
int operator <  (const String &s1, const String &s2)
int operator >  (const String &s1, const String &s2)
int operator <= (const String &s1, const String &s2)
int operator >= (const String &s1, const String &s2)
```

Die beiden Operanden werden jeweils wie mit `"strcmp"` verglichen (ASCII-Reihenfolge), und das Ergebnis ist ein boolscher Wert.

Ein Mini-Beispiel:

```
#include <tools/str.h>
#include <stream.h>

void main()
{ String s1 = "Test", s2 = "Text";

  if (s1 < s2)
    cout « s1;
  else
    cout « s2;
}
```

Ein kleiner Wermutstropfen: Vergleiche wie

```
String s1;

if(s1 == "Nanu?") ...
```

sind nicht erlaubt, da mehrdeutig! Schließlich gibt es sämtliche Vergleichsoperationen auch auf Zeigern (und insbesondere Zeichenketten), und so weiß der Compiler oben nicht, ob er `"s1"` nach `"char*"` konvertieren und diese Adresse dann mit der des Strings „Nanu?" vergleichen, oder ob er aus der konstanten Zeichenkette mittels Konstruktoraufruf ein temporäres `"String"`-Objekt erzeugen und darauf den String-Operator `"=="` anwenden soll.

So geht's aber:

```
if (s1 == String("Nanu?"))
```

### 2.5.5 Ein- und Ausgabe

In den obigen Beispielen wurden mehrfach Objekte der Klasse `"String"` ausgegeben. Das ist ohne weiteres möglich, denn die Operatoren wurden - wie im Tutorial unter Abschnitt 4.2.2.7 beschrieben - entsprechend zusätzlich überladen:

```
class ostream& operator « (ostream&, const String&)

class istream& operator » (istream&, String&)
```

Die Ein- und Ausgabe von `"String"`-Objekten funktioniert damit genau wie die von gewöhnlichen `"char*"`-Zeichenvektoren.

## 2.6 Interna von MaxonC++: <streamdefs.h>

Jeder hat so seine kleinen Geheimnisse - aber MaxonC++ verrät sie hemmungslos. Na ja, zumindest gibt es in der Datei „*<streamdefs.h>*“ einige Definitionen, die über die interne Organisation der Dateiverwaltung von MaxonC++ Aufschluß geben.

In „*<stdio.h>*“ wird der enorm wichtige Datentyp **"FILE"** definiert, der nichts als ein Alias für **"struct stream"** darstellt. Dieser Strukturtyp wird in „*<streamdefs.h>*“ definiert, und zwar wie folgt:

```
struct stream
{ unsigned Filehandle;
  char UngetCh, UngetBuf;
  signed char Mode, Error;
  struct streambuffer *bufptr;
  struct {
    int f_freemem:1,
    f_closefile:1
    } flags;
};
```

Die Felder haben folgende Bedeutung:

**Filehandle:** Dies ist die AmigaDOS-Filehandle, die zur Datei gehört.

**UngetCh:** Ein Flag, das anzeigt, ob ein Zeichen mit **"ungetc"** zurückgestellt wurde.

**UngetBuf:** Der Puffer, in dem eben jenes Zeichen abgelegt wird.

**Mode:** In diesem Byte steht ist das Bit #0 gesetzt, wenn die Datei zum Lesen geöffnet wurde, und Bit #1 steht entsprechend für Schreibzugriffe.

**bufptr:** Wenn dieser Zeiger nicht Null ist, zeigt er auf eine **"streambuffer"**-Struktur, in der Informationen über das Puffern der Datei enthalten sind.

**flags:** Das Bit **"f_freemem"** (Nummer 7) dieses Byte-großen Bitvektors zeigt an, ob **"fclose"** die **"stream"**-Struktur aus einer gewissen internen Liste aushängen soll, und **"f_closefile"** wird gesetzt, wenn bei **"fclose"** die DOS-Datei „*Filehandle*“ geschlossen werden soll.

Die Struktur **"streambuffer"** wird normalerweise von den Funktionen **"setvbuf"** oder **"setbuf"** erzeugt und in die **"stream"**-Struktur eingehängt. Sie ist folgendermaßen aufgebaut:

```
struct streambuffer
{ stream *streamptr;
  short size, fill, pos;
  signed char mode, own;
  int (*read)(register streambuffer *a0, register void *d2,
    register unsigned d3);
  int (*write)(register streambuffer *a0,
    register void *d2, register unsigned d3);
  int (*flush)(register streambuffer *a0);
```

```
    int (*close)(register streambuffer *a0);
    void *buf;
};
```

Die Bedeutung der Member:

**streamptr:** Ein Rückverweis auf die "stream"-Struktur, deren Zeiger "bufptr" wiederum auf diese "streambuffer"-Strukt ur zeigt.

**size:** Puffergröße (in Bytes)

**fill:** Anzahl der momentan benutzten Puffer-Bytes

**pos:** Bei Lesepuffern die aktuelle Leseposition

**mode:** Ist positiv, wenn es sich um einen Schreibpuffer handelt, negativ, wenn der Puffer zum Lesen dient, und Null, wenn der Puffer aus irgendwelchen Gründen überhaupt nicht benutzt werden soll.

**own:** Ein Flag, das gesetzt wird, wenn "setvbuf" den für den Puffer "buf" benötigten Speicher selbst alloziert hat. Dadurch weiß die Laufzeitbibliothek beim Schließen der Datei, ob der Speicherbereich, auf den "buf" zeigt, freigegeben werden muß.

**read:** Ein Zeiger auf eine Funktion mit den angegebenen Register-Parametern. Sie wird bei Eingaben aus der gepufferten Datei aufgerufen und sollte sich wie die AmigaDOS-Funktion "Read" verhalten, einmal davon abgesehen, daß das erste Argument hier keine Filehandle, sondern ein Zeiger auf den "streambuf" ist und auch nicht in "d1", sondern in "a0" übergeben wird.

**write:** Die entsprechende Funktion für Ausgaben in Dateien.

**flush:** Ebenfalls ein Zeiger auf eine Funktion. Sie wird (wie man sich denken kann) aufgerufen, wenn auf der Datei "fflush" ausgeführt wird oder wenn sie geschlossen wird.

**close:** Diese Funktion wird beim Schließen der Datei aufgerufen und sollte insbesondere den vom Puffer belegten Speicher freigeben. Falls die "streambuffer"-Struktur dynamisch erzeugt wurde, sollte die "close"-Funktion auch diese löschen.

**buf:** Ein Zeiger auf den eigentlichen Puffer.

Die Sache mit diesen Zeigern auf Funktionen hätte man natürlich viel eleganter und einfacher mit Vererbung und virtuellen Memberfunktionen realisieren können, aber die Definitionen sollten auch im ANSI C-Modus benutzt werden können. Es steht dem Programmierer natürlich frei, diesen Defi-

nitionen einen objektorientierten Ansatz zu überlagern, so wie in *„stream.h"* das C++-typische Stromkonzept auf die ANSI-Dateiverwaltung aufgesetzt wird.

Was soll das ganze eingentlich? Nun, mit diesen Informationen (und vielleicht ein wenig Herumprobieren) ist es möglich, eigene Routine auf tiefem Niveau in die Ein-/Ausgaberoutinen der Maxon C++ Laufzeitbibliothek zu integrieren. Es ist ja nicht gesagt, daß eine Struktur **"streambuffer"** immer nur einen Puffer repräsentiert. Beispielsweise könnte man hier Routinen einhängen, die Ein- und Ausgaben in Intuition-Windows machen, und dann z. B. mit **"printf"** in einen Dialogfenster schreiben. Der Phantasie sind hier fast keine Grenzen gesetzt.

Übrigens: Wenn hinter einem **"stream"** gar keine wirkliche DOS-Datei, sondern ein **"streambuffer"** mit selbstgeschriebenen geheimnisvollen Funktionen steht, sollte man den Member **"Filehandle"** auf Null setzen. Dann wissen Funktionen wie **"fseek"** oder **"ftell"** Bescheid und versuchen erst gar nicht, auf dieser nicht existenten DOS-Datei zu operieren.

Um das Herumtrashen an der Dateiverwaltung zu erleichtern, gibt es die Funktion **"allocstream"**:

```
stream *allocstream(unsigned Handle)
```

Sie erzeugt ein neues Objekt des Typs **"stream"**, trägt **"Handle"** als Filehandle ein und vermerkt die Struktur in der internen Liste der Dateien. Dadurch wird der Strom z. B. am Dateiende automatisch geschlossen, und **"fflush(0)"** wirkt auch auf diesen Stream. Dementsprechend wird auch das Flagbit **"f_freemem"** gesetzt.

Falls nicht mehr genug Speicher für die Struktur frei ist, liefert **"allocstream"** wie üblich eine Null.

## 2.6 Funktionen für originelle Linker-Optionen: <linkerfunc.h>

In der Include-Datei „*<linkerfunc.h>*" finden Sie einige MaxonC++-typische Funktionen, die Sie vor allem dann brauchen, wenn Sie ohne Startup-Code linken (siehe auch Abschnitt 4.6 des Benutzerhandbuchs).

Jede Objektdatei kann in C++ dynamische Initialisierungen benötigen, also eine Funktion besitzen, die beim Programmstart unbedingt aufgerufen werden muß. Außerdem kann es natürlich statische Daten geben, für die am Programmende Destruktoren aufgerufen werden müssen. Der Linker generiert Funktionen, in denen alle diese Initialisierungs- bzw. Aufräumfunktionen augerufen werden:

```
void InitModules()
```

ruft alle Initialisierungen auf, und

```
void CleanupModules()
```

die Destruktoren. Wenn man sich einen Startup-Code selbst schreibt, sollte man am Anfang auf jeden Fall `"InitModules"` und am Programmende `"CleanupModules"` aufrufen.

Es kommt bisweilen vor, daß eine C++-Funktion nicht aus einem C++-Programm, sondern von irgendwoher aufgerufen wird, z. B. wenn man eine Shared Library in C++ geschrieben hat. Das gibt natürlich Probleme, wenn man in seiner Funktion „Small Data„ verwendet, denn dann vertraut die Funktion darauf, daß im Register A4 ein Zeiger auf den Datenhunk steht.

Was tun? Nun, man muß diesen Zeiger explizit laden, indem man ganz am Anfang der Funktion (also, jedenfalls vor jeglichem Zugriff auf statische Daten)

```
void GetBaseReg()
```

aufruft.

# Maxon C++ 3

## Features

### Light-Version
- C++-Compiler (AT&T 2.0 Standard)
- ANSI C-Compiler (Aztec- und Lattice-ANSI-kompatibel)
- CLI-Compiler
- HotHelp 3-Reader zum Anzeigen von HotHelp-Projekten

### Developer-Version
- C++-Compiler (AT&T 3.0 Standard)
- Templates, Exceptions
- ANSI C-Compiler (wie Light-Version)
- CLI-Compiler
- beinhaltet zusätzlich: MaxonHotHelp 3, MaxonSDB, MaxonEASY-OBJECTS und MaxonASM-CLI

### Compiler (C/C++)
- komfortable Projektverwaltung (Make)
- direkter Fehlerstellenansprung
- voll in den Editor integriert
- über ARexx-Schnittstelle steuerbar
- sehr schnelle Turnaroundzeiten
- echte vorkompilierte Include-Dateien (Preincludes)
- Include-Dateien für Kickstart 3.1
- speicherbarer Programmstatus
- Codegenerator für prozessorspezifischen Code 68010/20/30 und 68881/2 FPU

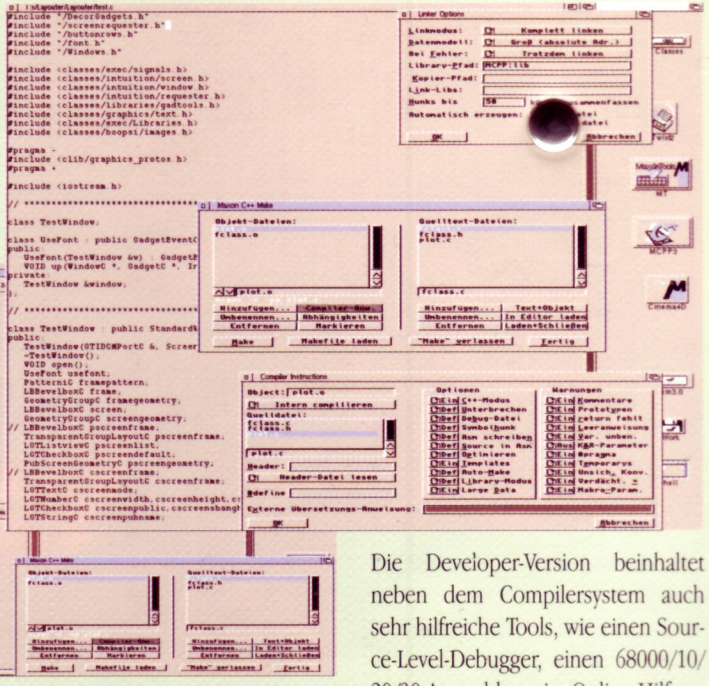
### Programmeditor (EDWARD)
- Multi-Window-Technik
- beliebig viele Texte gleichzeitig bearbeitbar (speicherabhängig)
- Tastatur- und Menüfunktionen frei belegbar
- speicherbarer Programmstatus
- ARexx-Schnittstelle

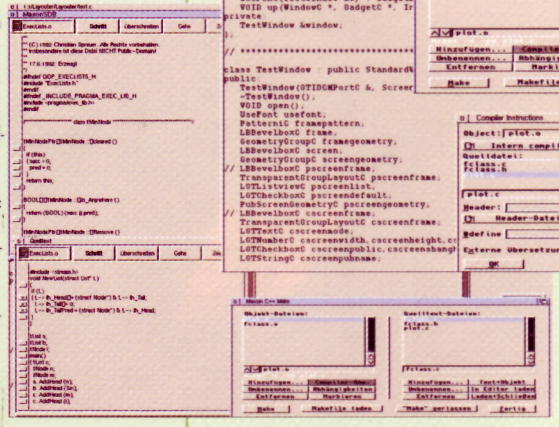
### Hilfesystem HotHelp 3
**(nur in der Developer-Version)**
- Deutschsprachige Beschreibung der Library- und Device-Funktionen zu Amiga-OS 2.1
- jederzeit aktivierbar
- kontextsensitiv
- komfortabler Editor für eigene Projekte
- Online-Projekte (direkte übernahme von Texten und Sourcecode in das Hilfesystem)
- ARexx-Schnittstelle
- Grafikeinbindung direkt im Hilfetext
- direktes Lesen von AmigaGuide™-Projekten
- lizenzfreie Weitergabe des HotHelp 3-Anzeigers

### Features der Klassenbibliothek (MaxonEASY-OBJECTS)
**(nur in der Developer-Version)**
- Ressourcenhandling: Betriebssystemressourcen (Speicher, Signale, Fenster, Bildschirme etc) werden automatisch verwaltet und bei Programmende freigegeben (keine aufwendige Fehlerbehandlung mehr nötig).
- Fehlerbehandlung durch Exceptions
- Handlerkonzept
- Datenstrukturen: Listen, dynamisch große Arrays, Datenpuffer (z.B. für I/O-Operationen)
- Verwaltung folgender Betiebssystemresourcen: Exec: Signale, Nachrichtenports, Speicher, Libraries, Devices, Listen - DOS: Locks, Dateien, Pfadnamen, Kommandozeilenargumente, Notifikation - Utility: Hooks und Taglisten - Intuition: Bildschirme, Fenster, IDCMP-Nachrichten, Systemrequester - Libraries: GadTools, ASL, Icons, Workbench, Tooltypes, Applikationenfenster
- BOOPSI Unterstützung
- automatisch größenveränderbare und font-veränderbare Fenster durch den Layouter
- deutschsprachige Online-Dokumentation

### Externer Assembler (MaxonASM-CLI)
**(nur in der Developer-Version)**
- 68000/10/20/30 Makro-Assembler
- FPU 68881/2 und MMU-Unterstützung
- Multi-Pass-Optimierung erzeugt sehr schnellen und kompakten Programmcode.
- voll in die Projektverwaltung integrierbar

### C/C++ Source Level Debugger (MaxonSDB)
**(nur in der Developer-Version)**
- Multi-Window-Technik
- einfache Bedienung - Drag&Drop
- Beobachten von Variableninhalten
- Haltepunkte
- direkt vom Compiler aus aufrufbar





**MaxonC++ ist die erste konsequente Implementierung des C++ 3 Standards auf dem Amiga.**

Das MaxonC++-Compilersystem beinhaltet sowohl einen ANSI C-, als auch einen zum AT&T 3.0-Standard kompatiblen C++-Compiler. Der Compiler erzeugt aus den C++-Quelltexten direkt ausführbare Programme ohne Umweg über C, Assembler oder andere Sprachen. Durch die volle Systemintegration werden überzeugend schnelle Turnaroundzeiten erreicht.

Die Developer-Version beinhaltet neben dem Compilersystem auch sehr hilfreiche Tools, wie einen Source-Level-Debugger, einen 68000/10/20/30-Assembler, ein Online-Hilfesystem mit der deutschsprachigen Beschreibung der Systemfunktionen und eine umfangreiche Klassenbibliothek.

Das auch in der Light-Version aufwendige und gut strukturierte deutschsprachige Handbuch ist ein kompletter ANSI-C/C++-Programmierlehrgang und auch für Einsteiger zu empfehlen.

Lauffähig auf allen Amigas mit min. 2 MB Speicher und Amiga-OS 2.0. Bei Nutzung der Klassenbibliothek werden 6 MB Speicher benötigt.

AMIGA®

4 010509 411302

MAXON Computer GmbH • Industriestr. 26
D-65760 Eschborn • Telefon 06196/481811
Irrtümer und Änderungen vorbehalten!

MAXON computer